

	Type	L #	Hits	Search Text	DBs	Time Stamp
1	BRS	L1	2571	transaction near order\$3	USPAT; US-PG PUB; EPO; JPO; DERW ENT; IBM_T DB	2004/01/28 13:49
2	BRS	L2	167	bus adj transaction near3 order\$3	USPAT; US-PG PUB; EPO; JPO; DERW ENT; IBM_T DB	2004/01/28 13:49
3	BRS	L3	127371 0	"in order"	USPAT; US-PG PUB; EPO; JPO; DERW ENT; IBM_T DB	2004/01/28 13:49
4	BRS	L4	0	"out of order"	USPAT; US-PG PUB; EPO; JPO; DERW ENT; IBM_T DB	2004/01/28 13:50
5	BRS	L5	29403	out adj order	USPAT; US-PG PUB; EPO; JPO; DERW ENT; IBM_T DB	2004/01/28 13:50
6	BRS	L6	12	2 and 3 and 5	USPAT; US-PG PUB; EPO; JPO; DERW ENT; IBM_T DB	2004/01/28 14:03

	Type	L #	Hits	Search Text	DBs	Time Stamp
7	BRS	L7	771	3 same 5	USPAT; US-PG PUB; EPO; JPO; DERW ENT; IBM_T DB	2004/01/28 14:17
8	BRS	L8	2	2 and 7	USPAT; US-PG PUB; EPO; JPO; DERW ENT; IBM_T DB	2004/01/28 14:12
9	BRS	L9	765	710/310,306,311.ccls.	USPAT; US-PG PUB; EPO; JPO; DERW ENT; IBM_T DB	2004/01/28 14:12
10	BRS	L10	2	7 and 9	USPAT; US-PG PUB; EPO; JPO; DERW ENT; IBM_T DB	2004/01/28 14:12
11	BRS	L11	19	("5440547" "5450411" "5513177" "5621898" "5640392" "5642349" "5742847" "5761430" "5790546" "5841771" "5875190" "5948080" "6026092" "6122279" "6145045" "6167471" "6233637" "6272131" "6272563" "2001/0014102").BN	USPAT	2004/01/28 14:14

	Type	L #	Hits	Search Text	DBs	Time Stamp
12	BRS	L12	60	("4464772" "4858116" "4941083" "4949239" "4992938" "5068781" "5086387" "5263172" "5307345" "5434996" "5440547" "5450411" "5471587" "5487170" "5513177" "5515376" "5546392" "5548736" "5621898" "5640392" "5640518" "5642349" "5742847" "5761430" "5761448" "5761450" "5761453" "5764968" "5790546" "5815734" "5841771" "5872998" "5875190" "5884050" "5892933" "5933607" "5948080" "5956509" "5974516" "5978878" "6006303" "6011916" "6026092" "6032211" "6092141" "6098110" "6119192" "6122279" "6145045" "6151651" "6167471" "6199132" "6233637	USPAT	2004/01/28 14:15
13	BRS	L13	54	(3 same 5).ti,ab.	USPAT ; US-PG PUB; EPO; JPO; DERW ENT; IBM_T DB	2004/01/28 14:17



US006012118A

United States Patent [19][11] **Patent Number:** **6,012,118**

Jayakumar et al.

[45] **Date of Patent:** **Jan. 4, 2000**

[54] **METHOD AND APPARATUS FOR PERFORMING BUS OPERATIONS IN A COMPUTER SYSTEM USING DEFERRED REPLIES RETURNED WITHOUT USING THE ADDRESS BUS**

[75] **Inventors:** **Muthurajan Jayakumar**, Sunnyvale, Calif.; **Sunny C. Huang**, Cupertino, both of Calif.; **Peter D. MacWilliams**, Aloha, Oreg.; **William S. Wu**, Cupertino, Calif.; **Stephen Pawlowski**, Beaverton, Oreg.; **Bindi A. Prasad**, Los Altos, Calif.

[73] **Assignee:** **Intel Corporation**, Santa Clara, Calif.

[21] **Appl. No.:** **08/954,442**

[22] **Filed:** **Oct. 20, 1997**

Related U.S. Application Data

[63] Continuation-in-part of application No. 08/774,512, Dec. 30, 1996, abandoned.

[51] **Int. Cl.⁷** **G06F 13/42**

[52] **U.S. Cl.** **710/126; 710/129; 710/107; 710/108; 710/112; 710/36; 710/39**

[58] **Field of Search** **395/280, 282, 395/285, 292, 309; 711/146; 710/36, 39, 105, 107, 129, 113, 243, 112, 108, 126**

[56] **References Cited****U.S. PATENT DOCUMENTS**

4,181,974	1/1980	Lemay et al. .
4,488,232	12/1984	Swaney et al. .
4,841,436	6/1989	Asano et al. .
5,006,982	4/1991	Ebersole et al. .
5,197,137	3/1993	Kumar et al. .
5,235,684	8/1993	Becker et al. .
5,615,343	3/1997	Sarangdhar et al. 395/282

FOREIGN PATENT DOCUMENTS

0275135 7/1988 European Pat. Off. .

OTHER PUBLICATIONS

European Search Report for Application No. EP94305316, 2 pages (Dec. 14, 1994).

Val Popescu et al., "The Metaflow Architecture," Metaflow Technologies, Inc., IEEE Micro, pp. 10-13 and 63-73 (Jun. 1991).

Primary Examiner—Ayaz R. Sheikh

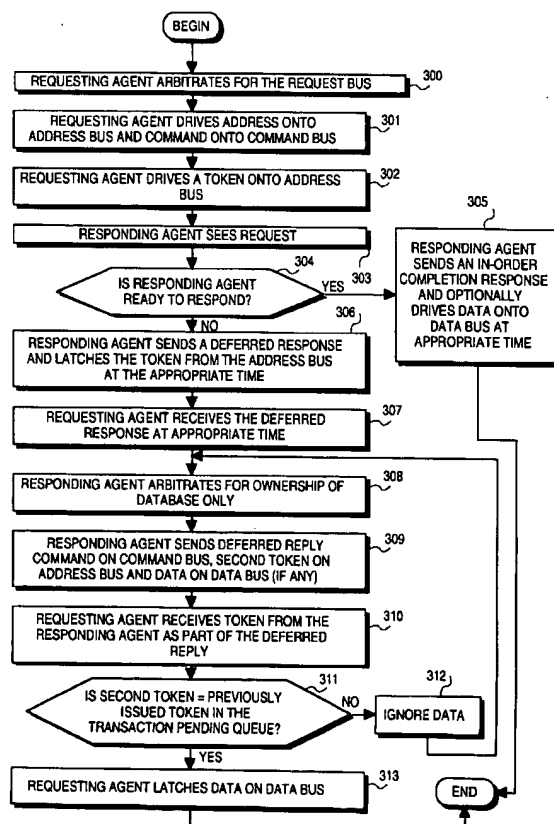
Assistant Examiner—Rupal D. Dharja

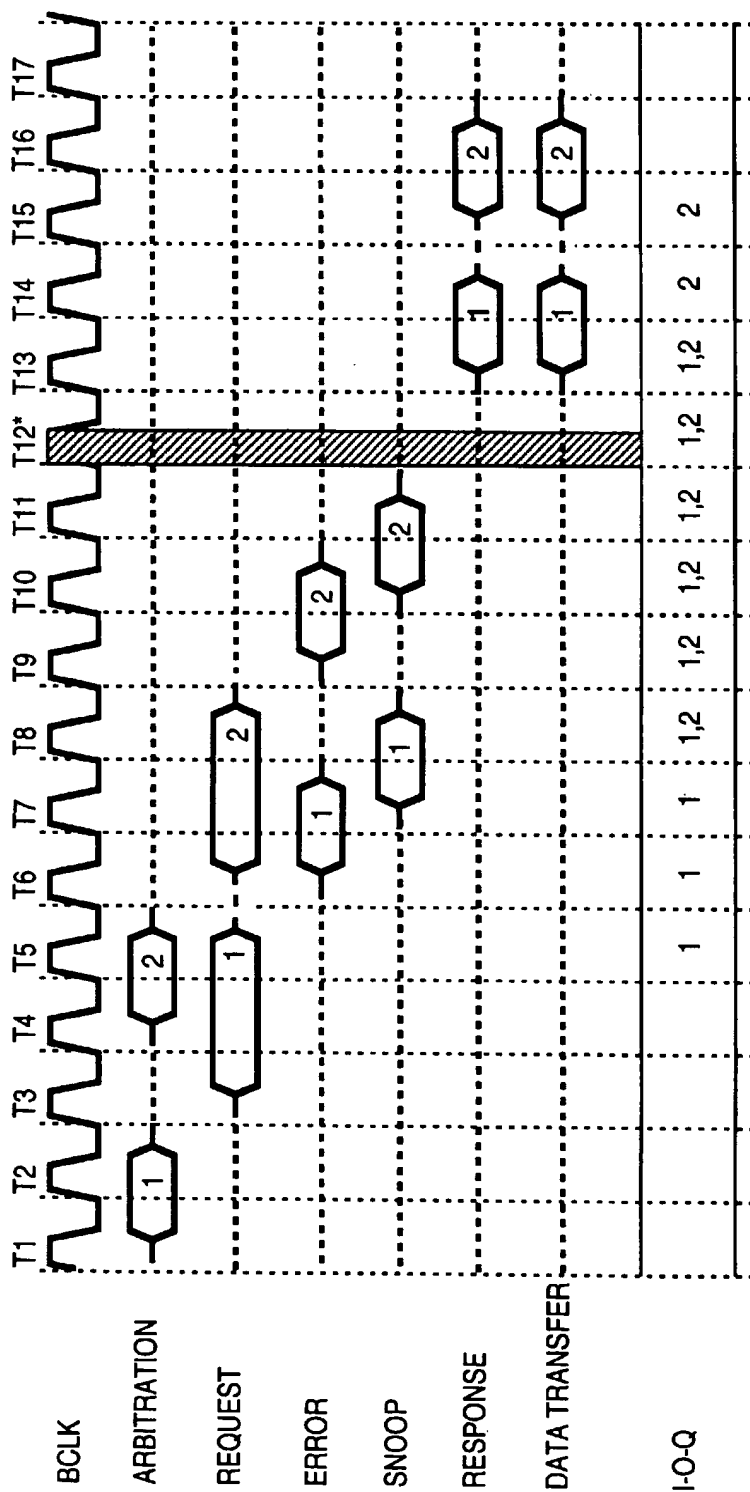
Attorney, Agent, or Firm—Blakely, Sokoloff, Taylor & Zafman LLP

[57] **ABSTRACT**

A split transaction bus in a computer system that permits out-of-order replies in a pipelined manner using an additional bus for use in the response phase.

39 Claims, 6 Drawing Sheets





*NOTE: THE SHADED VERTICAL BAR INDICATES ONE OR MORE CLOCK CYCLES ARE ALLOWED BETWEEN DIFFERENT PHASES.

FIG. 1
(PRIOR ART)

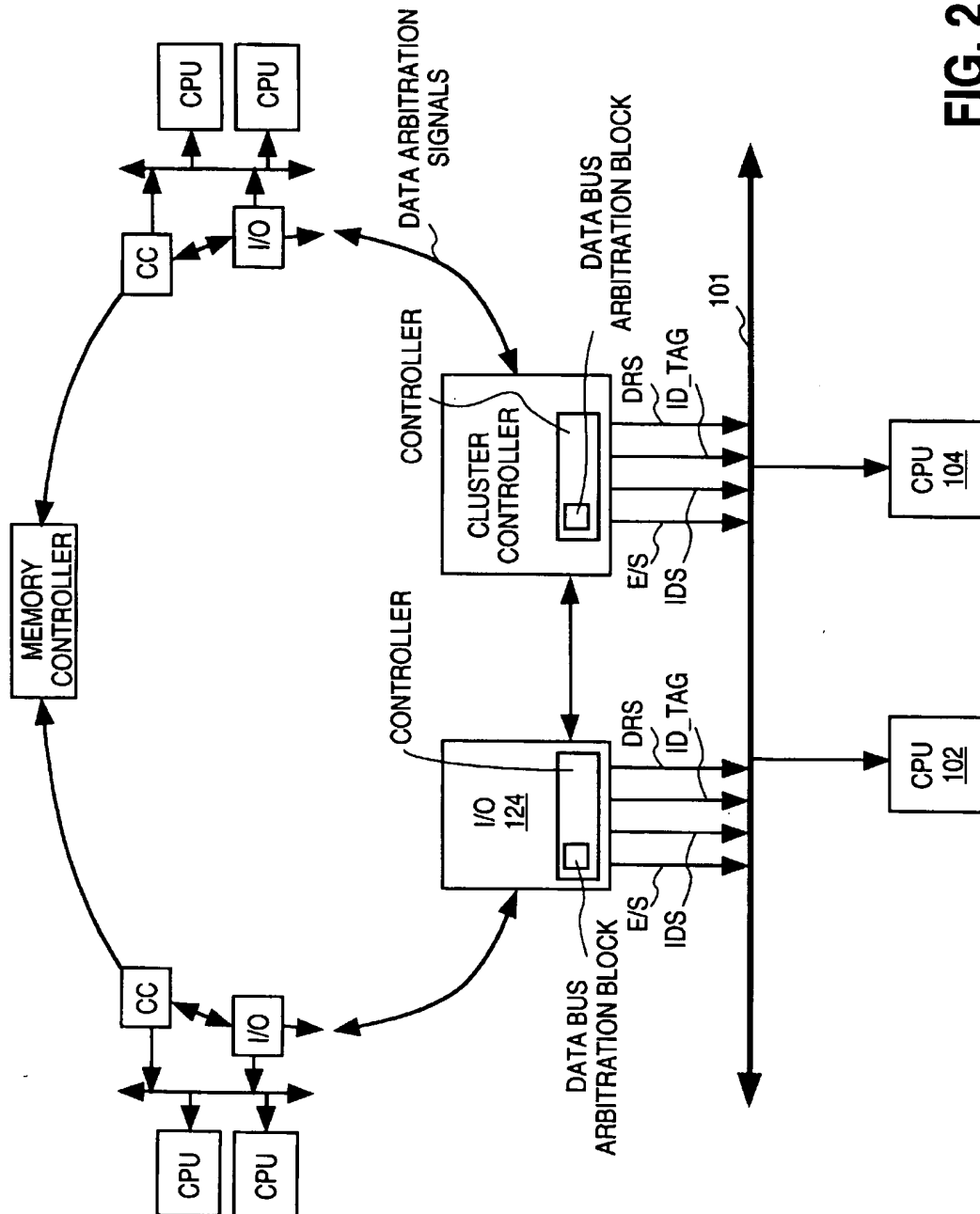
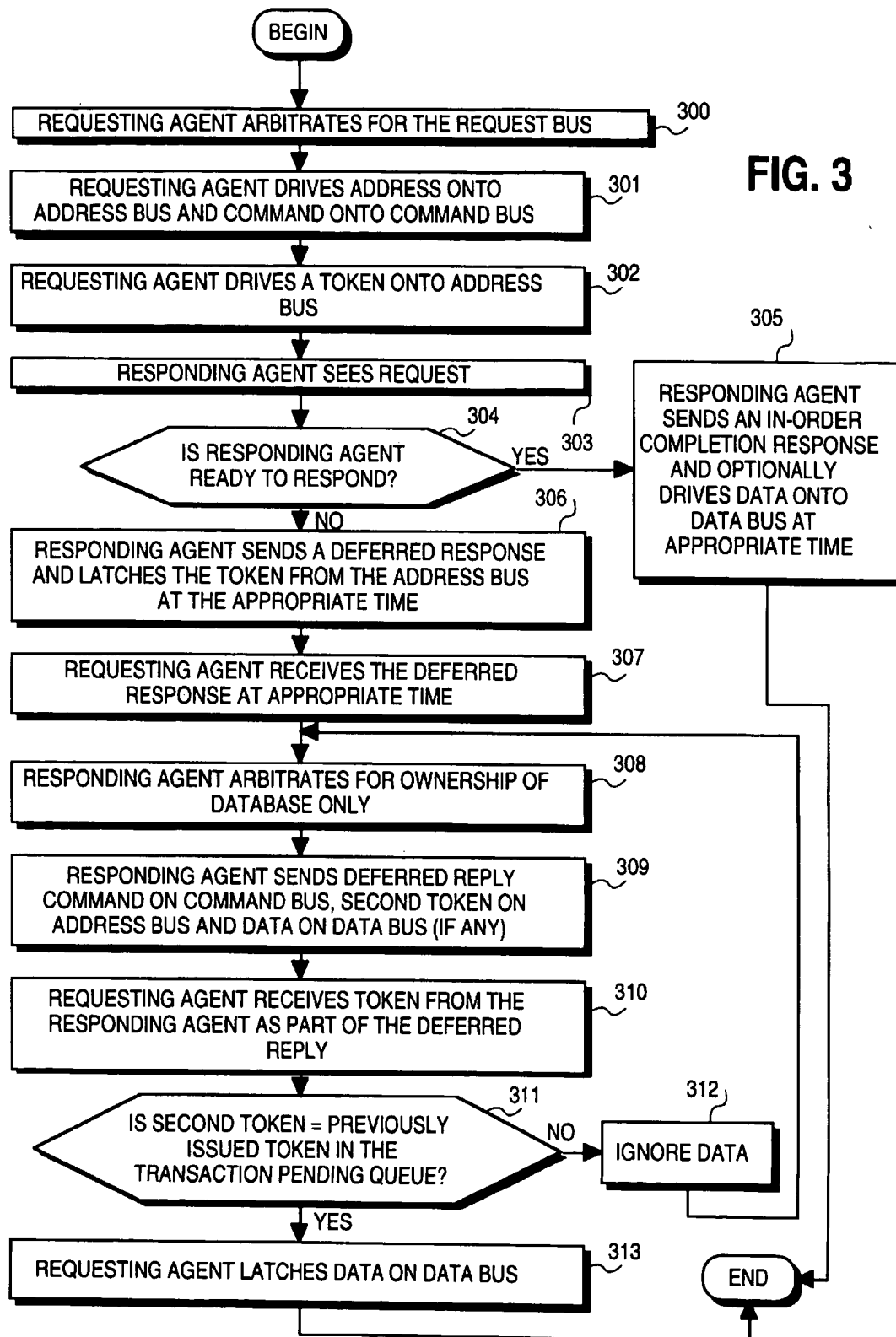


FIG. 2



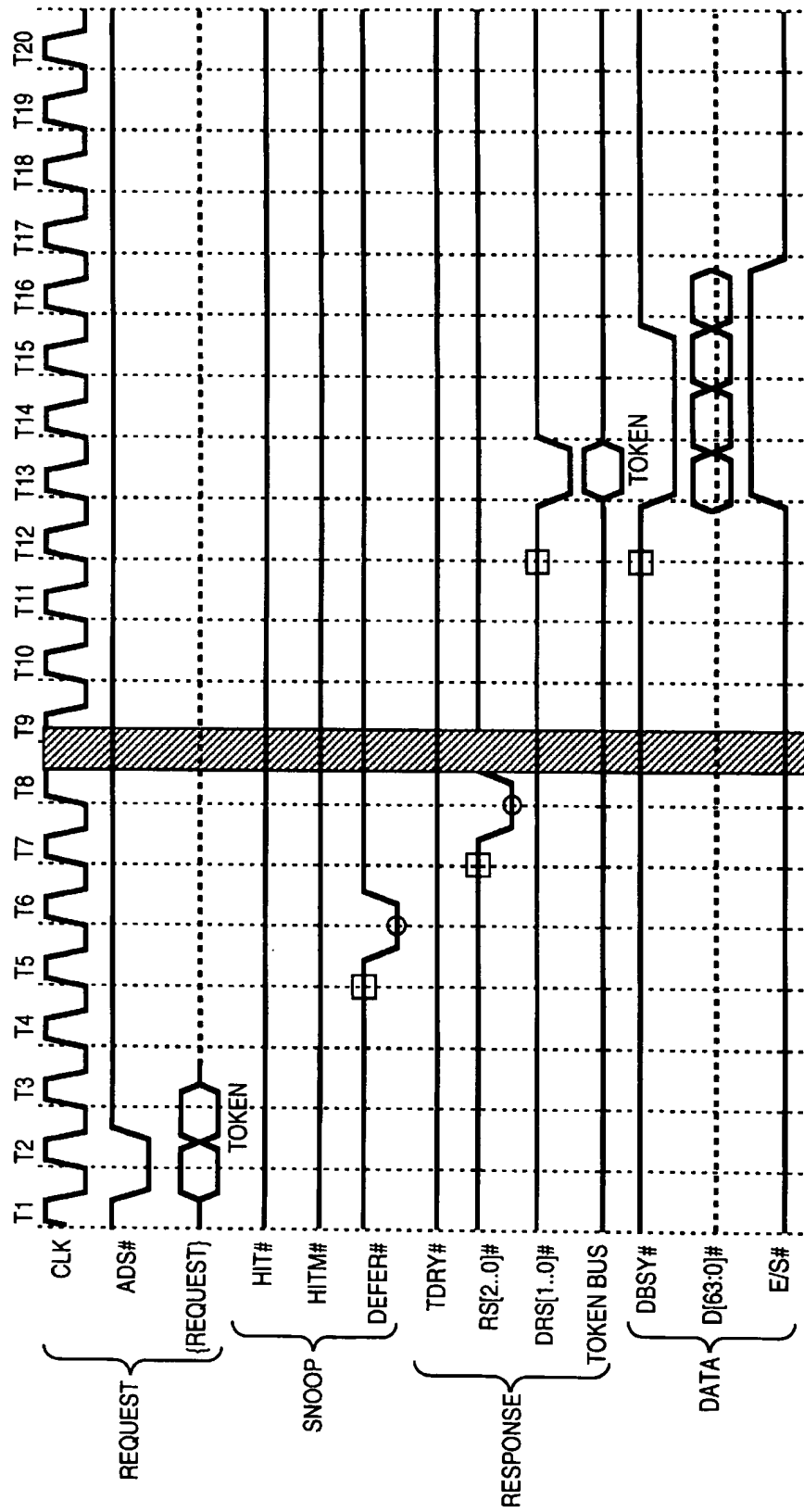
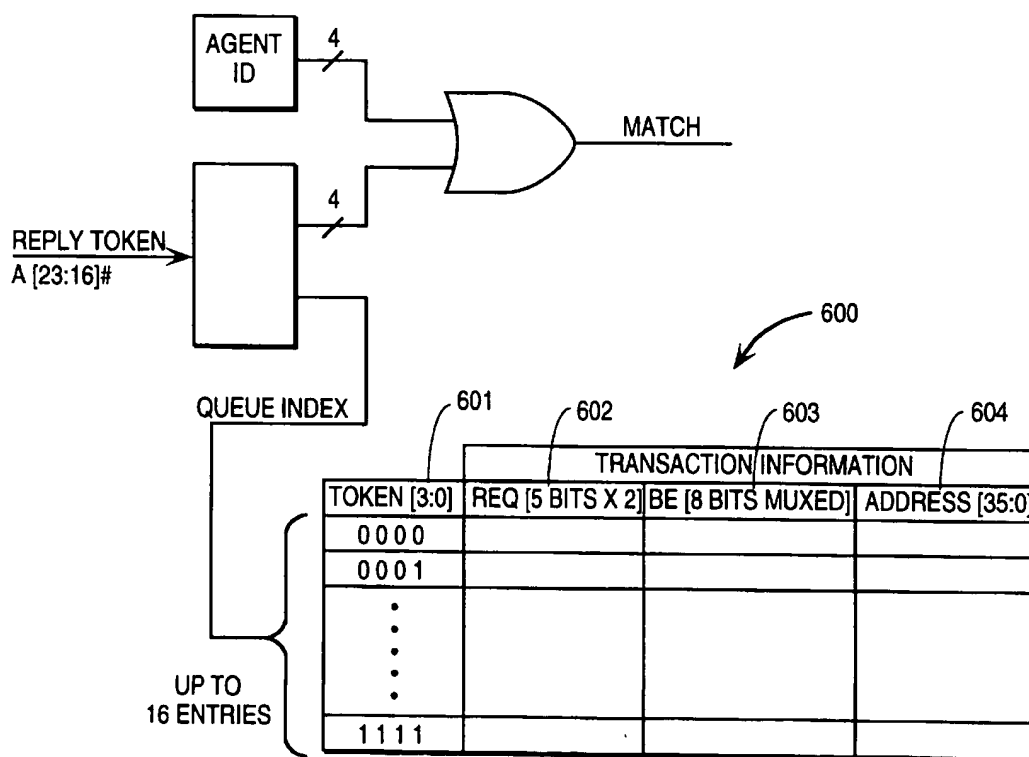
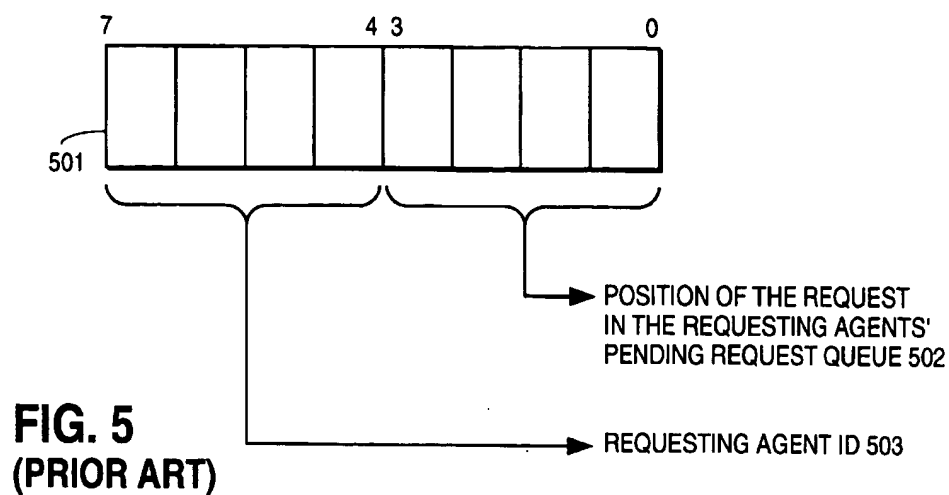


FIG. 4



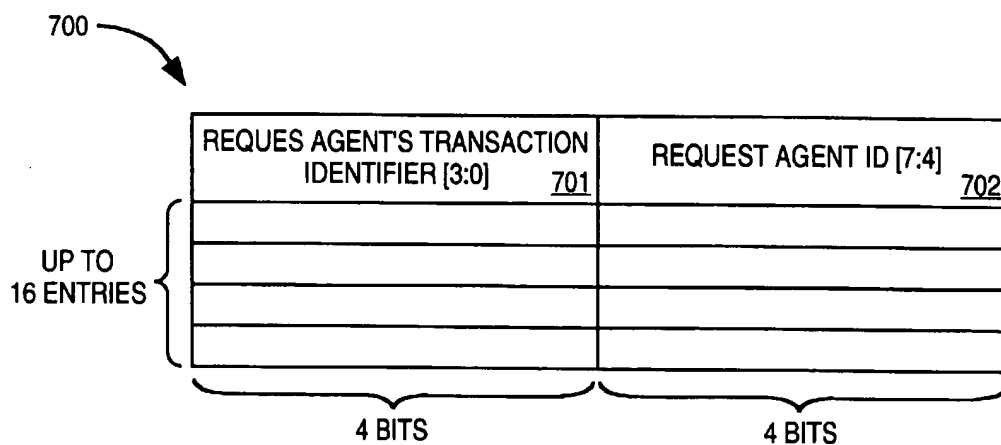


FIG. 7
(PRIOR ART)

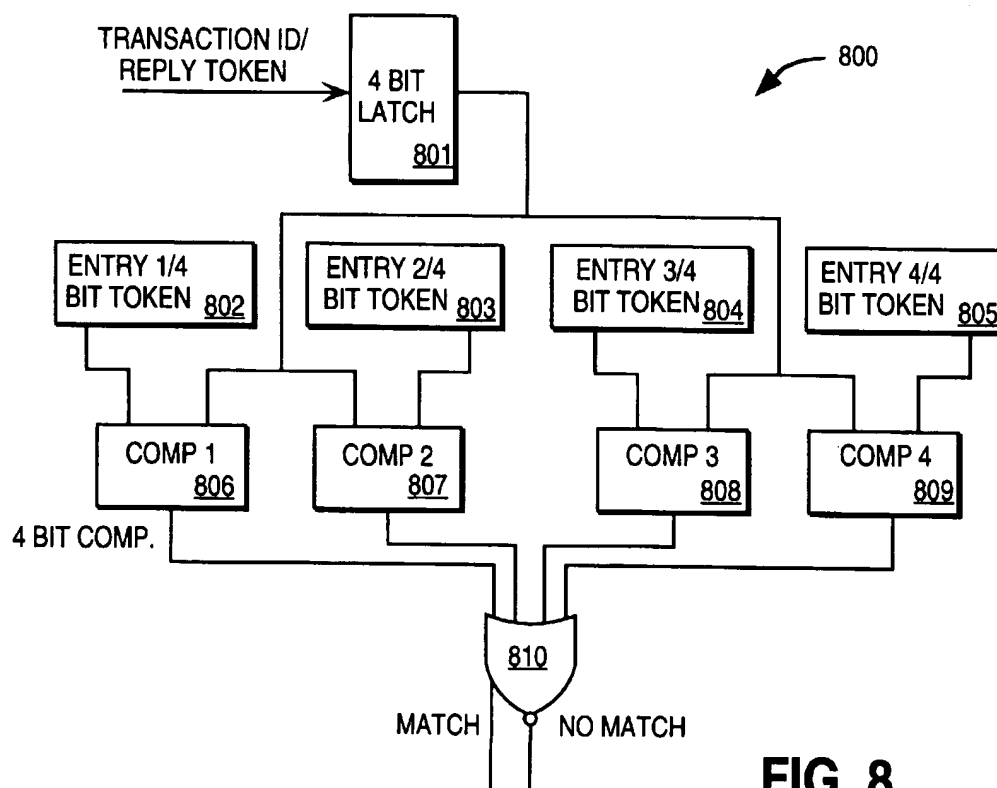


FIG. 8
(PRIOR ART)

METHOD AND APPARATUS FOR PERFORMING BUS OPERATIONS IN A COMPUTER SYSTEM USING DEFERRED REPLIES RETURNED WITHOUT USING THE ADDRESS BUS

This is a continuation-in-part of application Ser. No. 08,774,512, filed Dec. 30, 1996, now abandoned.

FIELD OF THE INVENTION

The invention relates to the field of protocols for computer system buses; particularly, the present invention relates to protocols for buses that perform deferred/out-of-order transactions.

BACKGROUND OF THE INVENTION

In a computer system, transfers between devices such as processors, memories, input/output (I/O) units and other peripherals generally occur according to a protocol. These devices are commonly referred to as agents. The protocol is a method of handshaking that occurs between the devices during a transfer which allows each device involved in the transfer to know how the other device is going to act or perform.

Typically, transfers of data and information in a computer system are performed using multiple buses. These buses may be dedicated buses between only two devices or non-dedicated buses that are used by a number of units, bus agents or devices. Moreover, buses in the system may be dedicated to transferring a specific type of information. For instance, an address bus is used to transfer addresses, while a data bus is used to transfer data.

A bus transaction normally includes a requesting device, or agent, requesting data or a completion signal from another agent on the bus. The request usually includes some number of control signals indicating the type of request accompanied by the address of the desired data or the desired device. The device which is mapped into the address space containing the requested address responds by sending a completion signal along with any data as necessary or after accepting the data.

In some computer systems, bus transactions occur in a pipelined manner. When bus transactions are pipelined, the requests from numerous bus agents are pending at the same time. This is possible due to the fact that separate data and address buses are used. In a pipelined transaction, while an address of a request is being sent on the address bus, the data or signals corresponding to a previously requested address (on the address bus) may be returned on the data bus. In certain pipelined systems, the completion responses occur in the same order as they were requested. However, in other pipelined systems, the order of the completion responses does not have to occur in the same order as their corresponding requests. This type of bus system is commonly referred to as a split transaction bus system.

In split transaction buses, a request is initiated with a first bus transaction to one of the agents in the computer system. If the responding agent cannot provide the response to the request at this time, the response corresponding to the request may be disassociated from the request. Eventually, when the response is ready, the response with optional data is returned to the requesting agent. The requests may be tagged so that they may be identified by the requesting agent upon their return.

To accommodate split transactions, the systems require some capability of associating a response with its address

(i.e., its request). One approach is to use two separate token buses and a deferred bus. When performing a request, an address is driven onto the address bus. At the same time, a token is driven on the first token bus. This token is associated with the address request. The token is received by the agent which is to respond to the address request (i.e., the responding agent). When the agent is able to respond, the responding agent drives the token on the second token bus and the appropriate response on the data bus.

Using two token buses increases the number of pins that are required to interface with the external bus. For tokens that are 8-bits in size, using two separate token buses requires an additional sixteen pins to be added to the computer system, as well as additional space allocated on the computer board for the token buses. Moreover, the pins used to support the token buses must also be added to every bus agent package in the system. An increase in the number of pins often equates to an increase in the cost of the package. Thus, the cost of integrated circuit components in the system increases. On the other hand, the increase in bandwidth due to permitting split transactions is significant due to the ability to reorder long latency transactions behind short latency transactions issued later. It is desirable to support split bus transactions without incurring all of the increased cost of modifying integrated circuit components and the increased number of pins required.

Another approach to accommodate split transactions is to use the address and data bus to pass tokens instead of two token buses. Such an approach is described in U.S. Pat. No. 5,568,620, where a token is sent on the data bus while the address is sent on the address bus during a bus request. The responding agent drives the token onto the address bus when generating the appropriate response. Another approach is to send the token on the address bus following an address, instead of using the address bus. Thereafter, the token returns on the address bus during the response. Thus, there are approaches that do not use additional token buses.

A bus is measured by its latency, bandwidth, and scalability. In a design, tradeoffs are made among these parameters, typically at the expense of one or more of the others. Split transactions affect the latency. For instance, an additional number of cycles (e.g., 6-8) need be taken when returning data for a split transaction. If a particular device already has a long latency, the overall latency of the system is even worse. If the latency is performance critical, performance degrades as use of split transactions increases in frequency. What is needed is a system that reduces the latency of the system so that a system using split cycles frequently is not penalized.

One problem associated with the approaches to split transactions without the use of token buses is that the address bus is used twice. The processor uses the address bus when it issues a bus cycle in the system and if the cycle is split. The address bus is used again by the system to return the token when the system returns the reply for the split cycle. Therefore, the more split cycles are being used, the more address bus bandwidth is dissipated. Needing more bandwidth may be disadvantageous in that the address bus may have to be operated at a higher speed, thereby making it difficult to be electrically functional in a common clock scheme.

Another problem with dissipating more address bus bandwidth is that it directly results in loss of scalability for a given frequency. For example, if a bus is designed to support four processors and if the split cycles use up address bandwidth, then the bus may only be able to support two processors.

The present invention provides a method and apparatus for implementing a bus protocol that accommodates split transactions. The bus protocol of the present invention improves the latency, bandwidth and scalability and avoids the problems discussed above.

SUMMARY OF THE INVENTION

A computer system that accommodates split bus transactions is described. The computer system of the present invention comprises a bus, a requesting agent and a responding agent. The bus includes a response bus and a data bus. The requesting agent is coupled to the bus and generates a bus operation. The responding agent is coupled to the bus and provides a deferral response onto the response bus if not ready to complete the bus operation and thereafter provides an out-of-order deferred reply when ready to complete the bus operation. The addition of one separate token bus improves concurrency and latency, as well as maximum bandwidth obtainable.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will be understood more fully from the detailed description given below and from the accompanying drawings of various embodiments of the invention, which, however, should not be taken to limit the invention to the specific embodiments, but are for explanation and understanding only.

FIG. 1 is a timing diagram of a bus transaction in the currently preferred embodiment.

FIG. 2 is a block diagram of one embodiment of the computer system of the present invention.

FIG. 3 is a flow diagram of a bus cycle according to the present invention.

FIG. 4 is a timing diagram of depicting a deferred response according to the present invention.

FIG. 5 illustrates an 8-bit token of the present invention.

FIG. 6 illustrates one embodiment of the transaction pending queue in the receiving agent.

FIG. 7 illustrates one embodiment of the transaction pending queue requesting agent.

FIG. 8 is a block diagram of one embodiment of the token matching hardware of the requesting agent.

DETAILED DESCRIPTION OF THE INVENTION

A method and apparatus for accommodating split transactions in a computer system is described. In the following detailed description of the present invention numerous specific details are set forth, such as token size, queue size, etc., in order to provide a thorough understanding of the present invention. However, it will be apparent to one skilled in the art that the present invention may be practiced without these specific details. In other instances well known methods, functions, components and procedures have not been described in detail to avoid obscuring the present invention. Overview of the Present Invention

The present invention provides a bus architecture that supports split transactions with improved latency, bandwidth and scalability. The present invention provides a zero latency reply for a split transaction.

The present invention allows split transactions for both memory and I/O operations. In one embodiment, only read responses are deferred, while write operations occur in order. The present invention only needs to arbitrate for use

of the data bus when the data associated with the read is ready. Once the use of the data bus has been granted, the reply is sent with request identification (token) on the ID bus to identify the request corresponding to the data on the data bus. The reply has zero latency because it does not require use of the address bus. Instead, the present invention uses a separate token, or ID, bus and a separate response phase for the zero latency reply in a split cycle. In one embodiment, the token comprises a deferred identifier (ID).

Two pins may be included for the new response group for split transactions. The two pins may be used to signal a deferred response, to an agent, that is being sent on the token bus and the data bus. The addition of the two pins allows the present invention to provide 300 Mbytes/sec bandwidth.

The present invention reduces the latency to zero cycles. Because of this reduction, a system is able to use split cycles more frequently without further being penalized. Also the present invention does not need the address bus to be used to transfer deferred replies, which results in a 50% savings in address bandwidth. Because of the reduction in address bandwidth, the speed demand on the address bus is also reduced, thereby reducing the risk in a common clock address bus scheme. Furthermore, since 50% of the address bandwidth is saved, as systems use have increased use of split cycles, there will be more scalability. Thus, the invention improves latency, bandwidth, scalability, and boosts the overall system performance.

The present invention is particularly advantageous to symmetric multi-processing which employs a distributed memory architecture. In one embodiment of the distributed memory architecture, all the memory cycles are split. The use of the present invention adds zero latency to these memory cycles, which enhances performance. Bus Transactions in the Present Invention

In the present invention, devices and units in the computer system perform bus transactions on the processor-memory bus. In one embodiment, the address bus and portions of the control bus of the processor-memory bus together function as a "request" bus for initiating and responding to requests. This "request" bus is used by the initiator of a request (e.g., a requesting agent) to transfer the request type and address and same bus is used by observers (e.g., responding agents) of the request to transfer their status or response for the issued transaction. The data bus transfers data being read or written. In another embodiment, the data bus is also used to transfer request identification information (e.g., a token).

In one embodiment, bus activity is hierarchically organized into operations, transactions, and phases. An operation is a bus procedure that appears atomic to software such as reading a naturally aligned memory location. Executing an operation usually requires one transaction but may require multiple transactions, such as in the case of deferred replies in which requests and replies are different transactions. A transaction is the set of bus activities related to a single request, from request bus arbitration through response-initiated data transfers on the data bus.

A transaction contains up to six phases. However, certain phases are optional based on the transaction and response type. A phase uses a particular signal group to communicate a particular type of information. In one embodiment, the six phases are:

- Arbitration
- Request
- Error
- Snoop
- Response

Data

The data phase is optional and used if a transaction transfers data on the data bus. The data phases are request-initiated, if the data is available at the time of initiating the request (e.g., for a write transaction), and response-initiated, if the data is not available at the time of generating the transaction response (e.g., for a read transaction).

Different phases from different transactions can overlap, thereby pipelining bus usage and improving bus performance. FIG. 1 is a timing diagram illustrating transaction phases for two transactions with data transfers. Referring to FIG. 1, when the requesting agent does not own the bus, transactions begin with an Arbitration Phase in which a requesting agent becomes the bus owner. However, in the present invention, when performing an out-of-order deferred reply, a responding agent may arbitrate for use of only the data bus without undergoing the arbitration phase which grants control of the entire bus (address, data, control buses). Thus, in the present invention, such a responding agent is allowed to proceed out-of-order with respect to the pipelined order on the address bus.

After the requesting agent becomes the bus owner, the transaction enters the Request Phase, in which the bus owner drives a request, and address information on the bus. The Request Phase is two clocks in duration. In the first clock, the ADS# signal is driven along with the transaction address and sufficient information to begin snooping and memory access. In the second clock, information used to identify the request (e.g., a token) and information about the length of the transaction are driven, along with other transaction information. In an alternate embodiment, the Request Phase is only one clock in duration and the request identification information is driven onto the data bus during the same clock as the ADS# signal is driven with the transaction address.

After the Request Phase, a new transaction enters a first-in-first-out (FIFO) queue, the In-Order queue. All bus agents, including the requesting agent, maintain an In-Order queue and add each new request to those queues. For more information on the in-order queue, see U.S. patent application Ser. No. 08/206,382 entitled "Highly Pipelined Bus Architecture", filed Mar. 1, 1994, assigned to the corporate assignee of the present invention and incorporated herein by reference. In FIG. 1, for example, request 1 is driven in T3, observed in T4, and in the In-Order beginning in T5.

The third phase of every transaction is an Error Phase, three clock cycles after the Request Phase. The Error Phase indicates any immediate (parity) errors triggered by the request.

If the transaction isn't canceled due to an error indicated in the Error Phase, a Snoop Phase is entered, four or more cycles from the Request Phase. The results of the Snoop Phase indicate if the address driven for a transaction references a valid or modified (dirty) cache line in any bus agent's cache. The Snoop Phase results also indicate whether a transaction will be completed in-order or may be deferred for possible out-of-order completion.

Transactions proceed through the In-Order Queue in FIFO order. The topmost transaction in the In-Order Queue enters the Response Phase. The Response Phase indicates whether the transaction has failed or succeeded, whether the transaction completion is immediate or deferred, whether the transaction will be retried and whether the transaction includes a Data Phase.

The valid transaction responses are: normal completion, hard failure, deferred, and retry.

If the transaction does not have a Data Phase, the transaction is complete after the Response Phase. If the request

agent has write data to transfer, or is requesting read data, the transaction has a Data Phase which may extend beyond the Response Phase.

In one embodiment, not all transactions contain all phases, not all phases occur in order, and some phases can be overlapped. All transactions that are not canceled in the Error phase have the Request, Error, Snoop, and Response Phases. Also, arbitration may be explicit or implicit in that the Arbitration Phase only needs to occur if the agent that is driving the next transaction does not already own the bus. The Data Phase only occurs if a transaction requires a data transfer. The Data Phase can be absent, response-initiated, request-initiated, snoop-initiated, or request- and snoop-initiated. The Response Phase overlaps the beginning of the Data Phase for read transactions (request-initiated data transfers or snoop-initiated data transfers). The Request Phase triggers the Data Phase for write transactions (request-initiated data transfers).

However, it should be noted that in the present invention the deferred reply is a separate bus transaction that is allowed to avoid the arbitration (of the address bus), request, error and snoop phases, proceeding directly to the response phase on read transactions, only arbitrating for use of the data bus to the send data.

Overview of the Computer System of the Present Invention
Referring first to FIG. 2, an overview of one embodiment of the computer system of the present invention is shown in block diagram form. It will be understood that while FIG. 2 is useful for providing an overall description of the computer system of the present invention, a number of details of the system are not shown. As necessary for disclosure of the present invention, further detail is set forth with reference to the other figures provided with this specification.

As illustrated in FIG. 2, in this embodiment, the computer system generally comprises of a processor-system bus or other communication means 101 for communicating information and processors 102 and 104 coupled with processor-system bus 101 for processing information. In the present invention, processor-system bus 101 includes subsidiary address, data and control buses. Processors 102 and 104 include an internal cache memory, commonly referred to as a level one (L1) cache memory for temporarily storing data and instructions on-chip. A level two (L2) cache memory may be coupled to either or both processors 102 and 104 for temporarily storing data and instructions for use by processor 102. In one embodiment, the L2 cache memory is included in the same chip package as processor 102 and/or processor 104.

A level three (L3) cache memory for temporarily storing data and instructions for use by other devices in the computer system (e.g., processor 102, etc.) and a L3 cache controller for controlling access to the L3 cache memory may also be coupled to processor-system bus 101.

A cluster controller 122 is coupled with processor-system bus 101 for controlling access to a random access memory (RAM) or other dynamic storage device, commonly referred to as a main memory for storing information and instructions for processors 102 and 104. These memories are well-known in the art and have not been shown to avoid obscuring the present invention.

An input/output (I/O) bridge 124 is also coupled to processor-system bus 101 to provide a communication path or gateway for devices on either processor-system bus 101 or the remainder of the system to access or transfer data between devices or other clusters on the other bus. I/O bridge 124 performs functions such as turning the byte/word/double-word data transfer traffic from I/O bus into line size traffic on processor-system bus 101.

I/O buses may be included to communicate information between devices in the computer system. Such buses, including their interconnection to components in the system, are well-known in the art. Devices that may be (optionally) coupled to these buses in the system include a display device, such as a cathode ray tube, liquid crystal display, etc., for displaying information to the computer user, an alphanumeric input device including alphanumeric and other keys, etc., for communicating information and command selections to other devices in the computer system (e.g., processor 102) and a cursor control device for controlling cursor movement. Moreover, a hard copy device, such as a plotter or printer, for providing a visual representation of the computer images and a mass storage device, such as a magnetic disk and disk drive, for storing information and instructions may also be coupled to I/O bus 131. These have not been shown to avoid obscuring the present invention.

The I/O bridge 124, cluster controller 132, processors 102 and 104 and the processor system bus 101 form a cluster. As shown, the computer system comprises multiple clusters. However, a system may only have one cluster, and the one or more clusters may include one or more processors. Each of the clusters, as well as each of the individual devices in the computer system, may be coupled by one or more buses. In one embodiment, each of the devices in the computer system is coupled to a network, mesh, monolithic interconnect (e.g., fabric), etc., which allows the devices to communicate. In one embodiment, this interconnection allows the devices to be approximately equal distance from each other.

Cluster controller 122 and I/O bridge 130 in each cluster are coupled to the I/O bus via address, data and control buses. Only pertinent portions of these buses have been shown to avoid obscuring the present invention. Each I/O bridge and cluster controller may comprise controllers such as bus controllers to support the split transaction of the present invention. Each of the I/O bridge and cluster controllers may also include a data bus arbitration block to perform data bus arbitration to gain access to the data bus when performing a response or a deferred reply according to the present invention. Such arbitration is well-known in the art. FIG. 1 illustrates the data arbitration signals between each of the I/O bridges and cluster controllers as well as the memory controller 140. These signals permit for data bus arbitration to be handled between the individual devices. In an alternative embodiment, a single data bus arbitration unit may be included in the system to accommodate all data bus arbitration for individual clusters. Note that allowing individual devices to support arbitration among themselves allows a bus protocol to be independent of the individual devices that are utilizing the deferred transactions of the present invention.

The cluster controller 122 and the I/O bridge 130 include, in part, the following pins:

DRS[1:0]#—Defer reply response used as the response status signals of when defer reply occurs. In one embodiment, the bus has a 2-bit encoding for 3 states: wait, retry, normal completion, and hardfailure. In one embodiment, "00" indicates that the device is not to generate out of order replies because the system cannot support them.

ID_tag#[7:0]—This tag (token) bus contains the transaction ID and the requesting agent ID that was sent to the cluster controller by the processor (or other requesting agent) that issued the request along with the address bus. In one embodiment, this bus only carries valid information if the DRS [1:0]# signals change state (when the defer reply is

happening in the split transaction). In one embodiment, a separate signal may be used to indicate that this token bus contains valid information (i.e., indicates the DRS [1:0]# signals have changed state). Such a signal may be the IDS# signal shown in FIG. 1.

E/S—The cluster controller indicates the state change for the cross-cluster snoop. In one embodiment, the E/S signal being high indicates an exclusive state, while the E/S signal being low indicates a shared state.

Using these signals, the controller is able to request an out-of-order defer reply through the DRS and the reply directly enters into the response phase of a transaction while arbitrating only for the data bus (not the address bus). That is, the controller is able to send the data onto the data bus without going through the arbitration, request, error, and snoop phases that are used when performing a bus transaction. Although there is arbitration for the data bus, such arbitration may be inherent by the nature of the pipelined bus. That is, because the bus is pipelined, multiple transactions are pending on the bus at the same time. In one embodiment, the processor acts as a slave device in that the decision of whether data may be transmitted to or from the processor under the control of the cluster controller or I/O bridge (or other bus master) and not under the control of the processor. In one embodiment, in the case of a read operation, the data is not sent to the processor until a DRDY# signal is asserted by another device on the bus. Similarly for a write transaction, the processor is not allowed to write data onto the bus until a TRDY# signal is asserted by another device on the bus. In both these cases, the DRDY# and TRDY# signals are coupled between the processor and the bus master(s). (Note that each of these signals as inputs to the processors may be shared by multiple devices.) Therefore, the data bus is under the control of the I/O bridges and cluster controllers. Thus, although the arbitration must occur for the data bus, there is no penalty associated with that arbitration because the I/O bridge and cluster controller are already in control of the data bus and know its availability as the data becomes available. Thus, in summary, although the control for the data bus is decided through arbitration, the deferred replies of the present invention may be used by directly going into the response phase and avoiding the typical arbitration, request, error and snoop phases.

Although in one embodiment of the present invention arbitration for the data bus is implicit, the present systems may be applied for use with arbitration is explicit. Furthermore, as discussed above, arbitration to resolve contention between two or more devices seeking control of the data bus is handled between the responding devices themselves, to the exclusion of the processors (i.e., the requesting devices).

Of course, other embodiments of the present invention may not require nor include all of the above components, while other embodiments require or include additional components. For example, the computer system may include an additional processor coupled to the processor-memory bus 101. The additional processor may comprise a parallel processor, such as a processor similar to or the same as processor 102, or may comprise a co-processor, such as a digital signal processor. In such embodiments, processor 102 and the additional processor(s) reside directly on a processor-memory bus 101.

Bus Protocol of the Present Invention

In the present invention, operations are performed on the processor-memory bus according to a bus protocol. The devices and units in the computer system of the present

invention represent both requesting agents and responding agents. A requesting agent is a device which is capable of requesting a bus operation that involves receipt of one or more completion signals and, optionally, data from another device. A responding agent is a device which is capable of responding to the request by sending the necessary signals and, optionally, data in response to the request. Responding agents that are able to accommodate split transactions are referred to herein as "smart" responding agents, while responding agents that cannot accommodate split transactions are referred to herein as "dumb" responding agents. Note that in the following description, the terms "agent" and "device" may be used interchangeably.

In the present invention, bus transactions occur on the buses in the computer system in a pipelined manner. That is, multiple bus transactions may be pending at the same time, wherein each is not fully completed. Therefore, when a requesting agent begins a bus transaction, the bus transaction may only be one of a number of bus transactions currently pending. Although bus transactions are pipelined, the bus transactions in the present invention do not have to be fully completed in order. The present invention allows for some of the completion replies to requests to be out-of-order by supporting deferred transactions. These are referred to as split transactions. Non-split transactions are completed in order.

The present invention accommodates deferred transactions by essentially splitting a bus operation into two independent bus transactions. The first bus transaction involves a request (for data or completion signals) by a requesting agent, and a response by the responding agent (or another agent). The request may include sending an address on the address bus. The request may also include the sending of a token, which the requesting agent uses to identify its request. The response may include the sending of completion signals (and possibly data) if the responding agent is ready to respond. In this case, the bus transaction ends, as well as the bus operation. However, if the responding agent is not ready to respond to the request (e.g., the completion signals and/or the optional data are not ready), the response may include providing an indication to the requesting agent that its response to the request will be deferred. This indication will be referred to herein as a deferral response.

In case of a deferral response, the bus operation comprises a second transaction that includes the responding agent sending a deferred reply with the completion signals and requested data (if any) to the requesting agent, such that the requesting agent completes the transaction. However, in the case of read operations, this second transaction differs from the first transaction because it only has a response phase. That is, the second transaction does not have an arbitration phase, a request phase, a error phase or a snoop phase as discussed above. The data is driven onto the data bus immediately after the responding agent is granted the use of the data bus, independent of the address bus or any other part of the request bus.

In one embodiment, the deferred reply includes completion signals (and possibly data) and comparison of request identifying information. The request identifying information comprises the token. That is, the token is resent by the responding agent to the requesting agent so that the requesting agent can associate the deferred reply with the original request. Deferred and non-deferred bus transactions according to one embodiment of the present invention are depicted in the flow chart of FIG. 3.

Referring to FIG. 3, a bus transaction, deferred or non-deferred, begins when a requesting agent arbitrates for

ownership of the address and control bus as used to make requests (processing block 300). Note that the control bus is used, in part, for the purposes of specifying commands, including those commands defining each bus transaction. In the present invention, the requesting agent arbitrates for the address bus and drives addresses and control signals to be used to specify a command. Upon winning the arbitration, the requesting agent drives a request in the form of an address onto the address bus (processing block 301) and the requisite control signals in a manner well-known in the art, and a first token onto the address bus (processing block 302). In one embodiment, the address and the first token are driven on the bus in two consecutive clocks. The token is stored in a queue, referred to as a pending request queue, with an indication of the request with which it is associated.

Once a request has been placed onto the bus, a determination of which device is to be the responding agent occurs (processing block 303). This determination includes a responding agent recognizing the address that was driven onto the address bus. In one embodiment, the responding agent is the device which is mapped into the address space that includes the address of the request.

The responding agent then determines if it is ready to respond (processing block 304). In case of a read operation, the responding agent is ready to respond if the data requested is available. In case of a write operation, the responding agent is ready to respond if it begins completing the write operation upon receipt, or, if capable of a post write, the responding agent is ready to respond as soon as the data and address are received (where the actual writing of the data is finished later). If the responding agent is ready to respond, then the responding device sends an "in-order" completion response indication and drives the necessary signals or data on the data bus at the appropriate time (processing block 205), thereby ending the bus transaction, and the bus operation as non-deferred.

If the responding agent is not ready to complete the bus transaction, then the responding agent latches the first token from the address bus and sends a deferral response at its appropriate response time (processing block 306). The responding agent may not be ready to respond if the data is not ready by the time the responding agent is required to respond in case of a read operation. The responding agent may not be ready to respond until a write operation has been completely finished, such as in the case of a write to an I/O device.

The requesting agent receives the deferral response on the bus at the appropriate time (processing block 307). The appropriate time is dictated by the pipelined nature of the system bus, which will be described in more detail below.

A number of additional bus transactions may be run on the bus after the requesting agent receives the deferral response before the responding agent is finally able to satisfy the request.

When the responding agent is ready to complete the deferred bus operation (e.g., the data does become available to the responding agent), the responding unit arbitrates for use of the data bus only (processing block 308), and the responding agent sends a deferred reply (processing block 309). Such arbitration would be well-known to one skilled in the art. As part of the deferred reply, the responding agent sends a second token on the token bus, a deferred reply signal (e.g., command) on the control bus and any appropriate data on the data bus. In one embodiment, the second token is the same as the first token; however, in other embodiments, the first and second tokens may not be the same, yet have a unique relationship to allow agents to correctly associate one with the other.

11

The requesting agent monitors the token bus, along with other agents, and receives the token (processing block 310). The requesting agent latches the second token. The requesting agent then determines whether the second token sent from the responding agent matches one of the outstanding tokens in its pending request queue (processing block 311). In such a case, the requesting agent performs a token match between the second token sent from the responding agent and the first token, or any other token stored in the requesting agent. It should be noted that in the present invention, the requesting agent is always monitoring the token bus for tokens.

If the requesting agent determines that second token from the responding agent does not match the first token, then the data on the data bus and the completion signal(s) are ignored (processing block 312). Note that in one embodiment, the first and second tokens are the same. If the requesting agent determines that second token from the responding agent matches the first token, then the deferred reply data on the data bus and/or the completion signals is the data or completion signals originally requested by the requesting agent and the requesting agent receives the completion signals and/or latches the data on the data bus (processing block 313), thus completing the second bus transaction. After receiving the completion signals and/or data corresponding to the original request, the entire bus operation ends. Thus, the bus transaction occurs between the responding agent and the requesting agent and the original bus operation is completed.

After a device has been determined to be the responding agent, the responding agent is responsible for generating a response to ensure that bus retains its pipelined nature. A responding agent may not be able to complete the requested bus transaction at the appropriate times. For example, the data requested may not be available. The data may not be available for numerous reasons. For example, the data may not be available due to long access times of a particular device. Also, the preparation of a response to satisfy the request may require a large number of cycles to complete due to the structure or the nature of a particular device.

Therefore, in the event that the responding agent cannot complete the requested bus transaction at the appropriate times, the responding agent sends a deferral response. This may be done using the RS[1:0] signals. Thus, the responding agent sends either the requested data or completion signals (if ready to respond) or a deferred completion indication comprising a deferral response (if unable to supply the requested completion signals or the requested data) at the appropriate time.

The appropriate time for a responding agent to generate a response is defined as that time at which the responding agent must respond with some type of response in order to retain the order of the responses on the data bus with those requests that occurred on the address bus, thereby ensuring the pipelined nature of the bus of the present invention. In other words, the order of the "responses" on the data bus must occur in the same order as the "requests" on the address bus. Therefore, some type of response to a particular request must appear on the data bus after the response of the request which immediately preceded the particular request on the address bus. If some type of response is not possible in a reasonable time (e.g., ten bus clocks), then all of the subsequent responses will be stalled by the agents responsible for generating them, hurting system performance. Note that the response itself may also be stalled.

To accommodate multiple pending requests that require deferred replies, the responding agent upon completing the bus cycle with a deferral response must be capable of

12

designating the deferred reply with a particular request if the responding agent is to be able to accommodate multiple requests while one or more requests are pending. For example, when data is requested from the cache memory (e.g., L3 cache memory) and the cache controller sends a deferral response, if the cache memory is to be accessed while this request is pending, the responding agent (e.g., the cache controller) must include a mechanism to associate the data that becomes available to the request which requested it. In one embodiment, in processing block 306 the responding agents store deferred requests (and associated tokens) for which a deferral response is generated. When the data is available, the responding agent is able to identify the destination of the data using the stored token and deferred request. The responding agent is responsible for maintaining consistency when it provides a deferral response, such as when there are multiple requests to the same cache line.

The token latched by the responding agent is used as a future reference between the requesting or responding agent to identify the response to the requesting agent when made by the responding agent. The requesting agent using the token stored in the transaction pending queue (with an indication of the request with which it was associated) watches for the response to the request. When the responding agent is ready to complete the bus transaction, the same token (used when sending the request) is sent on the address bus. In one embodiment, a special command is used to indicate to agents on the bus that the token bus has a valid token and the data on the data bus is associated with the token. Associated with the special command (e.g., control signal(s)) is the data (on the data bus) that was originally requested, if any, by the requesting agent. In this manner, the subsequently sent token reestablishes contact with the requesting agent.

Deferred Operations

During the Request Phase of a bus transaction, an agent may assert a Defer Enable (DEN#) signal to indicate if the transaction can be given a deferred response. Note that the "#" symbol indicates that the signal is active low. When the DEN# signal is deasserted, the transaction must not receive a deferred response. In one embodiment, all transactions in a bus-locked operation, deferred reply transactions, and writeback transactions cannot be deferred. Transaction-latency sensitive agents can also use the DEN# signal feature to guarantee transaction completion within a restricted latency.

A DEFER# signal in conjunction with a HITM# signal are used to determine when a response is to be deferred. The HITM# signal, when asserted, indicates the presence of modified data in the cache. In-order completion of a transaction is indicated by the DEFER# signal being deasserted or the HITM# signal being asserted during the SnooP Phase, followed by normal completion or implicit writeback response in the Response Phase. When the Defer Enable (DEN#) signal is inactive (and the HITM# signal deasserted), the transaction can be completed in-order or it can be retried, but not deferred. Thus, any bus agent incapable of supporting a deferred response may use this to be compatible, requiring that the transaction be completed in-order as described above or retried. A transaction can be retried when the DEFER# signal is asserted and the HITM# signal is non-asserted during the SnooP Phase followed by a retry response during the Response Phase. That is, a bus agent incapable of supporting a deferred response will provide a retry response if unable to provide the required response at the time of the Response Phase.

When Defer Enable (DEN#) is asserted, the transaction can be completed in-order, can be retried, or deferred. When

a transaction is to receive a deferred reply, the DEFER# signal is asserted (while the HITM# signal is deasserted) during the Snoop Phase followed by a deferral response in the Response Phase.

It is the responsibility of the agent addressed by the transaction to assert the DEFER# signal. In another embodiment, the responsibility can be given to a third party agent who can assert the DEFER# signal on behalf of the addressed agent.

In the present invention, a requesting agent assumes that every outstanding transaction issued with an asserted Defer Enable (DEN#) signal in the Request Phase may be completed with a deferred reply. To account for the possibility that each issued transaction may receive a deferred response, the requesting agent maintains an internal outstanding transaction queue, referred to herein as the pending request queue. Each entry in the queue has an ID. The number of queue entries (and the number of IDs) is equal to the number of requests the agent is able to issue and have pending. During the deferred reply of the second transaction, the requesting agent compares the reply sent on the address bus with all deferred IDs in its pending request queue. On an ID match, the requesting agent can retire the original transaction from its queue and complete the operation.

If a responding agent is unable to guarantee in-order completion of a transaction, the responding agent asserts a DEFER# signal in the transaction's Snoop Phase to indicate that the transaction is to receive a deferred reply or is to be aborted. If the DEFER# signal is not overridden by another caching agent asserting the HITM# signal (indicating a hit to a modified line in the cache) in the Snoop Phase, then the responding agent owns the transaction response. The response of the responding agent is determined by its reply policy. If the responding agent is capable of and desires completing the request in a subsequent independent bus transaction, then it generates a deferral response.

After issuing a deferral response and once ready to respond, the responding agent issues a second independent bus transaction to complete the earlier request for which the response was deferred. This second transaction may return data to complete an earlier transaction (e.g., in case of a read operation), or it may simply indicate the completion of an earlier transaction (e.g., in case of a write operation).

FIG. 4 illustrates a deferral response followed by the corresponding deferred response to a read operation with a deferred read out-of-order reply. Referring to FIG. 4, in T1, the requesting agent asserts the ADS# signal indicating that a valid bus definition and address is on the control and address buses. Also in T1, the requesting agent drives the {REQUEST} group to issue a Read Line request. In T2, the requesting agent drives the token into the address bus. The address and token are latched by the addressed agent in T2 and T3 respectively.

In the Snoop Phase, the addressed agent determines that the transaction cannot be completed in-order and hence asserts the DEFER# signal in T5. In T7, the addressed agent becomes the responding agent due to deasserted state of the HITM# signal in T6. The responding agent returns a deferral response in T7 by asserting the proper encoding on the RS[2:0]# signals. The RS[2:0]# signals are used to encode different types of responses, including a deferral response. Note that the responding agent stores the token latched from the address bus.

Prior to T12, the addressed responding agent obtains the data required in the original request. In T12, the responding agent issues a second transaction, by placing the token from the original transaction on the token bus and using the

DRS[2:0]# signals which indicate that the response is a deferred out-of-order replay. The requesting agent picks up snoop responsibility.

Also in T12, the responding agent drives normal completion response and begins the Data Phase, if necessary. The responding agent returns a successful completion response with data on the bus and drives data during T12-T15, which is received during T13-T16.

Note that in T13, the requesting agent observes the second transaction. The requesting agent compares the token received from the token bus with the token corresponding to the original request. In T16, the requesting agent receives the deferred reply and removes the transaction from its pending request queue (and the In-Order queue). This completes the bus operation.

In one embodiment, a memory agent or I/O agent in the computer system may defer a response on any request other than a bus locked request, another deferred reply transaction, or a cache line write designated for explicit writeback.

In the present invention, the deferred reply "wakes" up the original transaction. That is, the original transaction carries on and completes as if the requesting agent had initiated the transaction. For instance, when a memory agent returns a deferred reply to a processor in the system, the memory controller is not the master, and the processor acts as if it had initiated the transaction, responding to the handshaking on the bus.

Receiving a deferred reply removes the transaction from the In-Order queue. In order to complete the original request, the responding agent initiates a second independent transaction on the bus. The token (e.g., deferred reply ID), latched during the original transaction, is used as the address in the Request Phase of the second transaction.

In one embodiment, a token (e.g., deferred ID) contains eight bits, ID[7:0]#, divided into two four-bit fields. An exemplary token is shown in FIG. 5. The token is transferred on the ID bus (carrying ID[7:0]#) in the second clock of the original transaction's Request Phase. ID bus lines [7:4] contain the response agent ID (e.g., CPU ID), which is unique for every responding agent. ID bus lines [3:0] contain a request ID, assigned by the request agent based on its internal queue. An agent that supports more than sixteen outstanding deferred requests can use multiple agent IDs. In one embodiment, only four outstanding deferred requests may be supported by the processor of the computer system.

To complete the transaction, the ID bus is used to return the token, which was sent with the request on ID[7:0]#. During the Response Phase, the deferred reply is driven along with any data during the Data Phase. The requesting agent decodes the token returned on the token bus and compares it with previously deferred requests. The requesting agent compares the agent ID with its agent ID to see if there is a match. In case of a match, the requesting agent uses the remainder of the token as an index to point to the proper entry in its pending request queue. (Exemplary matching hardware is shown in FIG. 6.)

Responding Agent Embodiments

In the present invention, some responding agents never defer requests, some responding agents always defer requests, and some responding agents defer requests for certain operations and not other operations.

In the above illustrated embodiments, one agent may be the memory agent or I/O agent in a deferred operation.

If a caching memory controller, such as L3 cache controller 110, is in the computer system, the cache controller may also generate the deferred responses on read operations when the read data is not immediately available in the cache

memory. The L3 cache controller will then proceed to obtain the data from the memory, such as main memory 121, while servicing any subsequent requests for data available in the L3 cache memory. When the read response is ready, the L3 cache controller arbitrates for the bus and returns the read data.

If an I/O bridge is in the computer system, the I/O bridge may defer bus transactions. It should be noted that this is typically performed according to an I/O bus standard, such as the PCI, EISA, ISA, MCA, PCMCIA, VESA standard. Generally, accesses to devices residing on the I/O bus take longer to complete than accesses issued to the memory devices. In a highly pipelined bus involving requests generated from multiple requesting agents, there is an intermix of I/O operations and memory operations. In order to eliminate the stalling effect of I/O accesses in subsequent memory accesses, the I/O bridge of the present invention may generate a deferred response to accesses directed towards itself (and the devices on the I/O bus). In one embodiment, the I/O bridge of the present invention has the hardware necessary for deferring multiple bus requests. When the response is ready, the I/O bridge arbitrates for the bus and initiates a deferred reply (a second independent transaction).

Token Generation and Recognition Hardware

The bus architecture enables pipelined communication transactions wherein the differing bus phases of the pipelined transactions overlap. For one embodiment, the bus is configured to support up to eight outstanding communication transactions simultaneously. In one mode, each processor can issue up to four outstanding communication transactions on the bus. The pipelined communication transactions receive responses and data over the bus in the same order as the communication transactions are initiated on the bus.

Each bus agent coupled to the bus maintains internally a set of bus transaction queues that maintain transaction information in order to track the communication transactions. The transaction information includes the number of transactions outstanding, an identifier for the next transaction to receive a response, and a flag indicating the transactions issued by that bus agent.

Each bus agent coupled to the bus logs the bus transaction information for all transactions in an internal queue referred to as the in-order queue (IOQ).

All bus agents coupled to the bus maintain identical in-order queue status for tracking the communication transactions issued on the bus. Each communication transaction including deferred transactions issued on the bus is entered into the in-order queue of each bus agent coupled to the bus. For each bus agent, the communication transaction at the top of the in-order queue is the next transaction to enter the response and data phases.

In the present invention, each of the requesting and responding agents supports additional functionality in order to utilize the deferred protocol. In the present invention, the agents capable of deferral and deferred responses function as a bus master. Being a bus master according to the present invention comprises the ability to arbitrate for the bus and carry out a request transfer cycle. In the case of read transactions, these bus masters only arbitrate for the data bus. This allows the response agent to initiate the second transaction and to obtain control of the bus when completion signals or data is ready subsequent to the responding agent providing a deferral response. For instance, the processors are bus masters generally. On the other hand, memory controllers generally are not bus masters and, thus, would

need to have this ability to initiate the later transactions to complete the bus operation.

In the illustrated embodiment, token generation is the responsibility of the requesting agent. One embodiment of a token of the present invention is shown in FIG. 5. Referring to FIG. 5, token 501 is comprised of eight bits divided between two components: a position ID 502 indicating the position of the deferred request in the internal pending request queue of the requesting agent and a requesting agent ID 503 indicating the source of the token. (The pending request queue will be described in more detail below). The requesting agent ID 503 is comprised of four bits. The requesting agent ID 503 may be comprised of more or less than four bits. For instance, the requesting agent ID 503 may be a 3-bit ID. The number of bits used to represent the ID of the requesting agent indicates the maximum number of requesting agents in the computer system that may accommodate deferred responses. For example, by using four bits for the ID, sixteen requesting agents that generate deferrable requests may exist on the bus.

As shown in FIG. 5, the number of bits used to indicate the position of the request in the pending request queue of the requesting agent comprises of four bits. The position of the request may be described using more or less than four bits. For instance, the request position may be a 2-bit number. The number of bits used to describe the position of the request indicates the size of the request queue. For example, using four bits for the position request indicates that the request queue is capable of having a maximum of 16 separately addressable entries.

A responding agent that is capable of deferred responses maintains an internal deferred request queue for storing tokens latched from the address bus when providing a deferral response. In one embodiment, this queue contains 16 entries, but may have more or less (e.g., 2-32 entries). In the present invention, only a small queue depth is sufficient since the comparison hardware need support only the entries that are actually deferred.

One embodiment of a request queue in a responding agent is shown in FIG. 6. Referring to FIG. 6, request queue 600 is shown having multiple entries, wherein each entry includes multiple fields. In one embodiment, queue 600 is a 16 entry queue. In the present invention, each entry row has a token field 601, a request field (REQ) 602, a byte enable (BE) field 603 and an address field 604. REQ field 602, BE field 603 and address field 604 represent the transaction information corresponding to each token in token field 601 in queue 600. REQ field 602 uses two request fields to store the type of request. BE field 603 stores the byte enable information that is used in further qualifying the request. Address field 604 stores the address.

In one embodiment, token field 601 comprises four bits and represents an index, REQ field 602 comprises 5 bitsx2, BE field 603 comprises eight bits muxed, and address field 604 comprises 36 bits. The token field 601 may be implicit in the hardware and, hence, may not be physically implemented in the hardware. In one embodiment, the token field 601 is not implemented in hardware when the position of the entry in the queue is the same as the token field contents. Note also that the index of the token field may be a randomly generated value.

Also shown in FIG. 6 is comparison logic which receives four bits of the token (from a latch, for instance) representing the agent ID and compares them to the agent's ID to see if there is a match. In case of a match, the representing agent uses the other four bits as an index into the queue.

At the time the responding agent wishes to provide a deferral response, the responding agent assigns an entry for

the transaction in the deferred request queue for storing the token latched from the address bus. The assigned entry in the queue of the responding agent stores the token and the REQ, BE, and Address fields. After the transaction is complete, the responding agent becomes a request bus owner and initiates a second bus transaction to provide a deferred reply. The responding agent also reclaims free queue entries in the deferred request queue.

The requesting agents of the present invention include logic for managing the request pending queue and for token generation. Based on the above description, this logic is well within the ability of those skilled in the art, and thus, will not be described further. The responding agents also have logic for managing their queues. Preferably, the responding agent is able to take appropriate action on the new request of the bus while the deferred requests are being processed. The responding agent is further provided with the capability to compare the address of the new request against the contents of an entry in the deferred request queue. In one embodiment, each agent includes a latch and a comparator to compare the new request to entries stored in the deferred request queue of the responding agent. The size of the comparators may be reduced, such that only the lower bits in the address are compared. On a positive match, the responding agent may abort the new request to avoid any conflict with an already deferred request.

In the present invention, the requesting agents are capable of accepting and comprehending the deferred responses (e.g., the deferral response and the deferred reply). In the present invention, the requesting agents are also capable of taking the appropriate action(s) in response to these responses. One action may involve temporarily suspending a request.

In one embodiment, the requesting agents assume that every outstanding transaction issued in the Request Phase may receive a deferral response. Each requesting agent includes a queue to temporarily store deferred requests (e.g., tokens and their associated requests). Therefore, the requesting agents maintain an internal pending request queue with the same size as its ability to issue new requests. One embodiment of the pending request queue is shown in FIG. 7. Referring to FIG. 7, queue 700 is shown having four entries, where each entry is comprised of 8 bits which include a requesting agent's request identifier field 701 and a field for the requesting agent's ID 702. The requesting agent ID 702 is the same for all entries in a particular requesting agent's queue. The number of bits used to represent the identifiers limits the number of designed entries in queue 700. In one embodiment, the identifier comprises four bits, but may comprise as many bits as necessary to support the total number of entries in queue 700. The agent ID field 702 is comprised of 4-bits for storing the agent's ID. Together the request identifier 701 and agent ID 702 form the token.

On observing a deferral response, the requesting agent maintains a copy of the token provided with the request in the generated token field 702 in outstanding transaction queue 700. The requesting agents also have the ability to release the bus after it received deferral response.

In one embodiment, a separate request pending queue is not required when the bus agent already includes a register file or list of bus operations that are pending. Since the agent ID is common to all queue entries, only a storage area specifying the pending transactions is required. The In-order queue, or other memory storage, that maintains a record of operations for the bus can be used as a pending request queue. Each of the transactions are identified using an index

(e.g., 0,1,2,3). In one embodiment, only two bits are used to support index values for four possible pending transactions. A field for one or more bits may be added to the queue and used to indicate whether a request received a deferral response. Therefore, a separate request pending queue is not required. Note that token generation logic would have to obtain an agent's ID from a storage location and combine it with the request identifier (e.g., index into a bus queue) to generate a token.

The requesting agent also includes matching hardware for matching the subsequently returned data or completion signals to the appropriate request using the returned token. During the second independent transaction, the requesting agent matches the resent token with all stored tokens in its pending request queue. On an ID match, the requesting agent can complete the operation and discard the original suspended request from its pending request queue.

In one embodiment, each requesting agent includes comparator logic to perform the matching. In one embodiment, the comparator logic is comprised of 8-bit comparators for performing the matching and control logic for identifying the match and returning data to internal units with their interval tags. In another embodiment, separate agent ID and transaction ID (e.g., index) matching logic is used. One embodiment of the matching hardware for matching a transaction ID is shown in FIG. 8. Note that agent ID matching logic, such as described in FIG. 6 may be included as well. Referring to FIG. 8, matching hardware 800 comprises latch 801, latches 802-805 for storing the tokens (or a portion thereof) associated with deferred responses, comparators 806-809 for comparing the stored entries with the resent token, and NOR gate logic 810. Latch 801 is coupled to receive four bits of the reply token. The output of latch 801 is coupled to one input of each of comparators 806-809. The other input of each of the comparators 802-809 is coupled to the output of latches 802-805 respectively. The outputs of each of comparators 802-809 are coupled to the inputs of NOR gate logic 810.

In one embodiment, latch 801 comprises a 4-bit latch which receives the transaction ID from the resent token from the address bus. Latch 801 is sized according to the number of bits in the transaction ID of the resent token. Latch 801 supplies the transaction ID to each of comparators 802-809. Each of the tokens stored in the pending request queue of the requesting agent are supplied to the comparators 802-809 via latches 802-805. Latches 802-805 are sized according to the number of bits in the transaction ID of tokens.

After performing an agent ID match, each of comparators 802-809 compares the transaction ID of the resent token to those of the stored tokens. If there is a match to a token stored in the pending request queue, one of comparators 802-809 produces a high output. If there is not a match to a token stored in the pending request queue, none of comparators 802-809 produces a high output. If none of the outputs from comparators 802-809 is high, then the No Match output of NOR gate logic 810 is high, thereby indicating that the resent token does not correspond to a suspended request in the requesting agent. If one output from comparators 802-809 is high, then the match output of NOR gate logic 810 is high, thereby indicating that the resent token corresponds to a suspended request in the requesting agent. In case of a match, the requesting agent "wakes up" and latches the data on the data bus (optional) and receives the completion signals corresponding to the original request.

Note that latches and comparators large enough to accommodate the entire token (e.g., 8-bit comparators, latches)

19

may be used. However, in view of the fact that the agent ID portion of the resent token is always the same, complexity and cost may be reduced by providing only one agent ID comparison with separate transaction ID comparisons.

If a separate request pending queue is not required (and an existing bus queue is used), the matching logic of the present invention matches the agent ID portion of the resent token with its ID and then uses the remainder of the returned token as an index into the queue.

In an alternative embodiment, the present invention could use the request address as part or all of the token.

Whereas many alterations and modifications of the present invention will no doubt become apparent to a person of ordinary skill in the art after having read the foregoing description, it is to be understood that the particular embodiments shown and described by way of illustration is in no way intended to be considered limiting. Therefore, references to details of various embodiments are not intended to limit the scope of the claims which in themselves recite only those features regarded as essential to the invention.

Thus, a method and apparatus for performing deferred bus transactions in a computer system has been described.

We claim:

1. A computer system comprising:

a bus operable in a pipelined order and having a response bus, a data bus, an address bus, and a token bus;
a requesting agent coupled to the bus to generate a bus request; and

a responding agent coupled to the bus, wherein the responding agent provides a deferral response onto the response bus if not ready to complete the bus operation and thereafter provides an out-of-order deferred reply without using the address bus, the out-of-order deferred reply comprising request identification information on the token bus and data responsive to the request onto the data bus after arbitrating for use of the data bus only when ready to complete the bus operation.

2. The system defined in claim 1 wherein the responding agent provides the out-of-order deferred reply when the data requested by the bus operation is available.

3. The system defined in claim 1 wherein an out-of-order deferred reply indication is provided onto the response bus.

4. The system defined in claim 1 wherein the responding agent provides the deferral response in a response phase of a first bus transaction and provides the out-of-order deferred reply during a response phase of a second bus transaction.

5. The system defined in claim 4 wherein the responding agent initiates the second bus transaction and directly enters the response phase by asserting at least one signal on the response bus.

6. The system defined in claim 1 wherein the responding agent comprises an arbitration mechanism to arbitrate for control of the data bus independent of the address bus.

7. The system defined in claim 1 wherein the bus further comprises a request identification (ID) bus.

8. The system defined in claim 7 wherein the responding agent drives request ID information onto the request ID bus while driving the out-of-order deferred reply on the response bus.

9. The system defined in claim 8 wherein the responding unit provides an out-of-order deferred reply indication on the response bus, the request ID information onto the request ID bus, and the data onto the data bus in a response phase of a bus transaction.

10. A method for performing bus operations in a computer system, the method comprising steps of:

a first agent initiating a bus operation;

20

a second agent providing a deferral response to the first agent in response to the bus operation;

sending a deferred reply to the first agent without using an address bus including sending data onto a data bus after arbitrating for only the data bus;

the first agent receiving the deferred reply to complete the bus operation.

11. The method defined in claim 10 wherein the deferral response is sent as part of a first bus transaction and the out-of-order deferred reply is sent as part of a second bus transaction.

12. The method defined in claim 11 wherein second agent enters directly into a response phase of the second bus transaction when providing to out-of-order deferred reply.

13. The method defined in claim 10 further comprising the steps of:

retrieving data associated with the bus operation;

the second agent arbitrating for use of a data bus; and

the second agent sending the data on the data bus upon grant of ownership of the data bus to the second agent.

14. The method defined in claim 13 wherein the step of initiating a bus operation comprises the steps of:

the first agent driving an address on an address bus; and
the first agent driving request identification (ID) information on the address bus.

15. The method defined in claim 14 further comprising the step of sending the request ID information to the first agent on an ID bus.

16. A method for performing bus operations in a computer system, the method comprising steps of:

initiating a bus operation by driving an address on an address bus;

driving request identification (ID) information on the address bus;

receiving a response indicating that a deferred reply to the bus operation is to be received;

receiving the request ID information from an ID bus, data on a data bus and the deferred reply without using the address bus;

identifying the request ID information as corresponding to the bus operation; and

receiving the deferred reply to complete the bus operation.

17. The method defined in claim 16 wherein the step of receiving the request ID information and the deferred reply occurs after data associated with the bus operation is retrieved and an agent sending the data and the deferred reply arbitrates for use of a data bus.

18. The method defined in claim 17 further comprising arbitrating for use of the data bus occurs without arbitrating for use of the address bus.

19. The method defined in claim 18 wherein arbitrating for use of a data bus occurs as soon as the data is available.

20. The method defined in claim 16 further comprising the step of comparing the request ID information with a queue of pending transactions in the first agent and identifying the request ID information as corresponding to the bus operation if the request ID information matches one of the pending transactions in the queue.

21. The method defined in claim 16 wherein the request ID information comprises an agent ID.

22. The method defined in claim 21 wherein the agent ID comprises a processor ID.

23. The method defined in claim 16 wherein the request ID information comprises a storage location in a queue of pending transactions in the first agent.

21

24. The method defined in claim 16 wherein the step receiving the response comprises receiving an assertion of a defer signal.

25. The method defined in claim 16 further comprising receiving at least one completion signal to the first agent.

26. The method defined in claim 17 further comprising the step of driving data on a data bus in the computer system as part of the deferred reply.

27. A computer system comprising:

an address bus;

a request identification (ID) bus;

a requesting agent coupled to the address and request ID buses, wherein the requesting agent is operable to generate a bus request to start a first bus transaction by providing an address and request ID information on the address bus;

a responding agent coupled to the address and request ID buses, wherein said responding agent, responsive to the address and the request ID information, provides a response to the requesting agent indicating that a deferred reply is to be made to satisfy the bus request when the responding agent is not ready to complete the bus request, thereby completing the first bus transaction, and further wherein the responding agent drives request ID information on the request ID bus, data on the data bus, and the deferred reply as a second bus transaction independent of the address bus availability and without the address bus when ready to send the data to complete the bus request.

28. The system defined in claim 27 further comprising a data bus coupled to the requesting agent and the responding agent, wherein the responding agent retrieves data responsive to the bus request, arbitrates for use of the data bus, and sends the data on the data bus to the requesting agent as part of the second bus transaction upon grant of ownership of the data bus.

29. The system defined in claim 28 wherein the responding agent arbitrates for use of the data bus without arbitrating for use of the address bus.

30. The system defined in claim 28 wherein the responding agent arbitrates for use of the data bus as soon as the data is available.

31. The system defined in claim 27 wherein the requesting agent identifies the request ID information as corresponding to the bus request and accepts the deferred reply to complete the second transaction.

22

32. The system defined in claim 27 wherein the request ID bus comprises a uni-directional bus from the requesting agent to the responding agent.

33. The system defined in claim 27 wherein the requesting agent provides the address and the request ID information on the address bus in consecutive clock cycles.

34. The system defined in claim 27 wherein the responding agent is not ready to respond because the data corresponding to the bus request is not available.

35. A method for performing bus transactions in a computer system, the method comprising steps of:

receiving an address and request identification (ID) information on an address bus as part of a request to initiate a first bus transaction;

sending a deferral response in response to the request to complete the first bus transaction;

the second agent sending both the request identification information on an ID bus and a deferred reply as part of a second bus transaction without using the address bus;

the second agent arbitrating for use of a data bus; and the second agent sending the data on the data bus as part of the second bus transaction without using the address bus upon grant of ownership of the data bus to the second agent.

36. The method defined in claim 35 further comprising the steps of:

identifying the deferred reply as corresponding to the request; and

receiving the deferred reply to complete the second bus transaction, such that the request is completed in two bus transactions.

37. The method defined in claim 36 wherein the second agent arbitrates for use of a data bus occurs without arbitrating for use of the address bus.

38. The method defined in claim 36 wherein second agent arbitrates for use of a data bus occurs as soon as the data is available.

39. The method defined in claim 35 wherein the step of identifying includes comparing the request ID information with a plurality of pending bus operations in a queue in the first agent, and receiving the deferred reply if the request ID information matches one of the plurality of pending bus operations.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

PATENT NO. : 6,012,118

DATED : January 4, 2000

INVENTOR(S) : Jayakumar et al.

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

In column 18, at line 43, delete "802" and insert --806--.

In column 18, at line 45, delete "802" and insert --806--.

In column 18, at line 49, delete "802" and insert --806--.

In column 18, at line 52, delete "802" and insert --806--.

In column 18, at line 54, delete "802" and insert --806--.

In column 18, at line 55, delete "802" and insert --806--.

In column 18, at line 59, delete "802" and insert --806--.

Signed and Sealed this

Third Day of April, 2001

Attest:

Nicholas P. Godici

NICHOLAS P. GODICI

Attesting Officer

Acting Director of the United States Patent and Trademark Office



US005535345A

United States Patent [19]

Fisch et al.

[11] **Patent Number:** 5,535,345[45] **Date of Patent:** Jul. 9, 1996

[54] **METHOD AND APPARATUS FOR SEQUENCING MISALIGNED EXTERNAL BUS TRANSACTIONS IN WHICH THE ORDER OF COMPLETION OF CORRESPONDING SPLIT TRANSACTION REQUESTS IS GUARANTEED**

[75] Inventors: **Matthew A. Fisch; James M. Brayton**, both of Beaverton; **Ajay Malhotra**, Portland, all of Oreg.

[73] Assignee: **Intel Corporation**, Santa Clara, Calif.

[21] Appl. No.: **241,964**

[22] Filed: **May 12, 1994**

[51] Int. Cl.⁶ **H01J 13/00**

[52] U.S. Cl. **395/309; 395/288**

[58] Field of Search **395/287, 292, 395/288, 308, 307, 467, 436, 309, 306**

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,785,394	11/1988	Fischer	364/DIG. 1
4,860,198	8/1989	Takenaka	364/DIG. 1
5,191,649	3/1993	Cadambi et al.	395/200
5,297,242	3/1994	Miki	395/425
5,333,296	7/1994	Bouchard et al.	395/425
5,345,569	9/1994	Tran	395/375

OTHER PUBLICATIONS

"The Metaflow Architecture", pp. 10-13 and 63-73, by Val Popescu, Merle Schultz, John Spracklen, Gary Gibson, Bruce Lightner, and David Isaman, IEEE Micro, 1991.

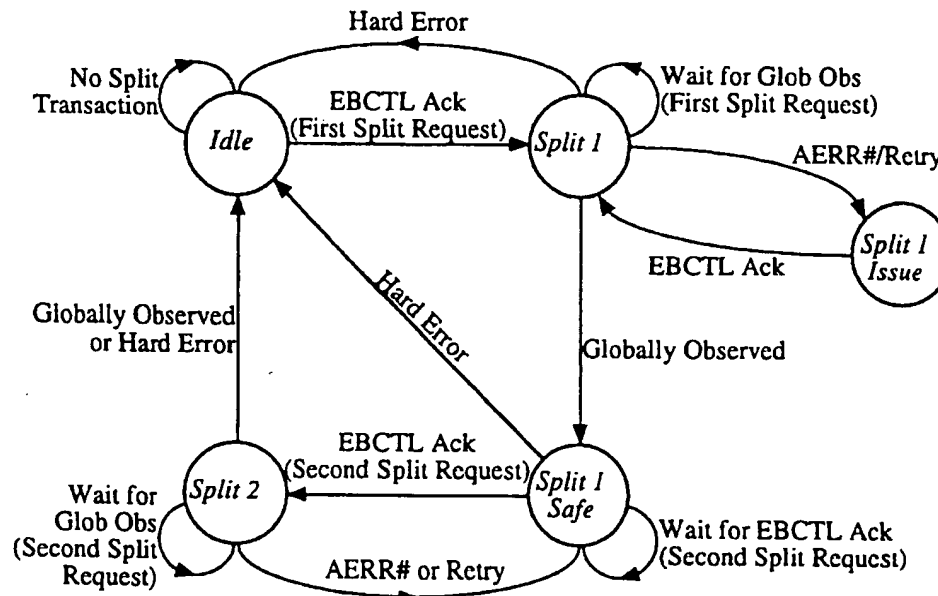
Primary Examiner—Jack B. Harvey

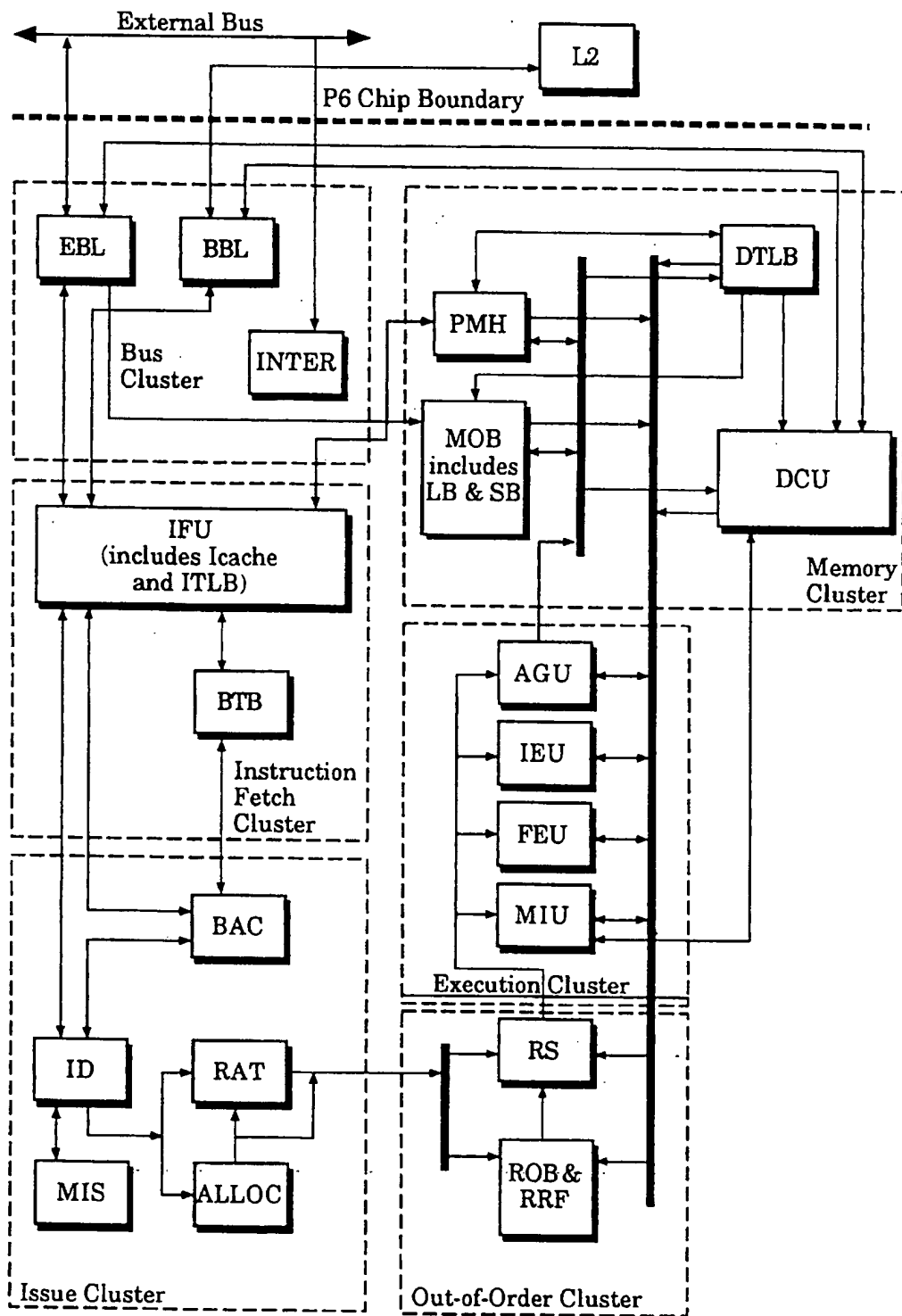
Assistant Examiner—Xuong M. Chung-Trans
Attorney, Agent, or Firm—Blakely, Sokoloff, Taylor & Zafman

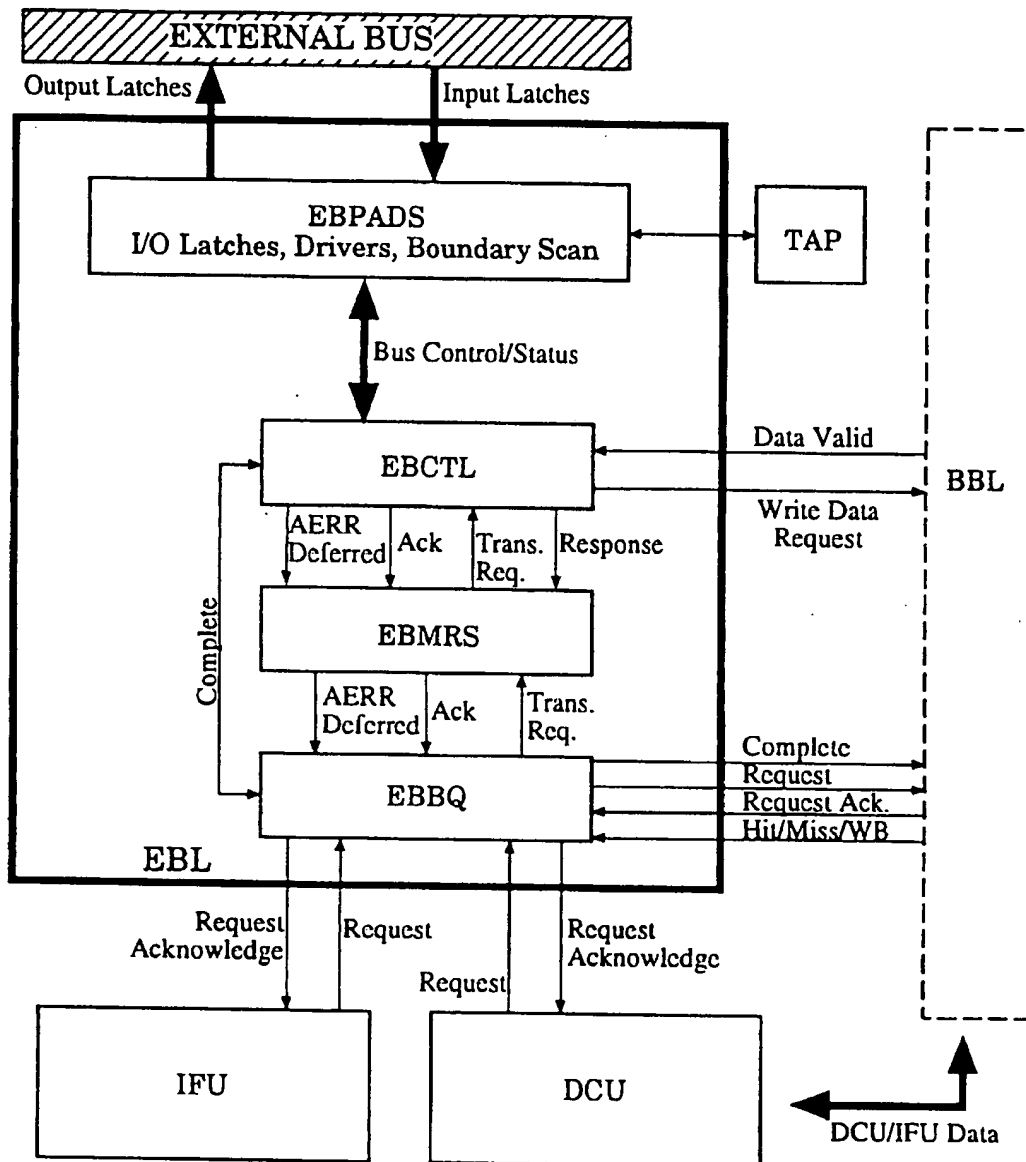
[57] **ABSTRACT**

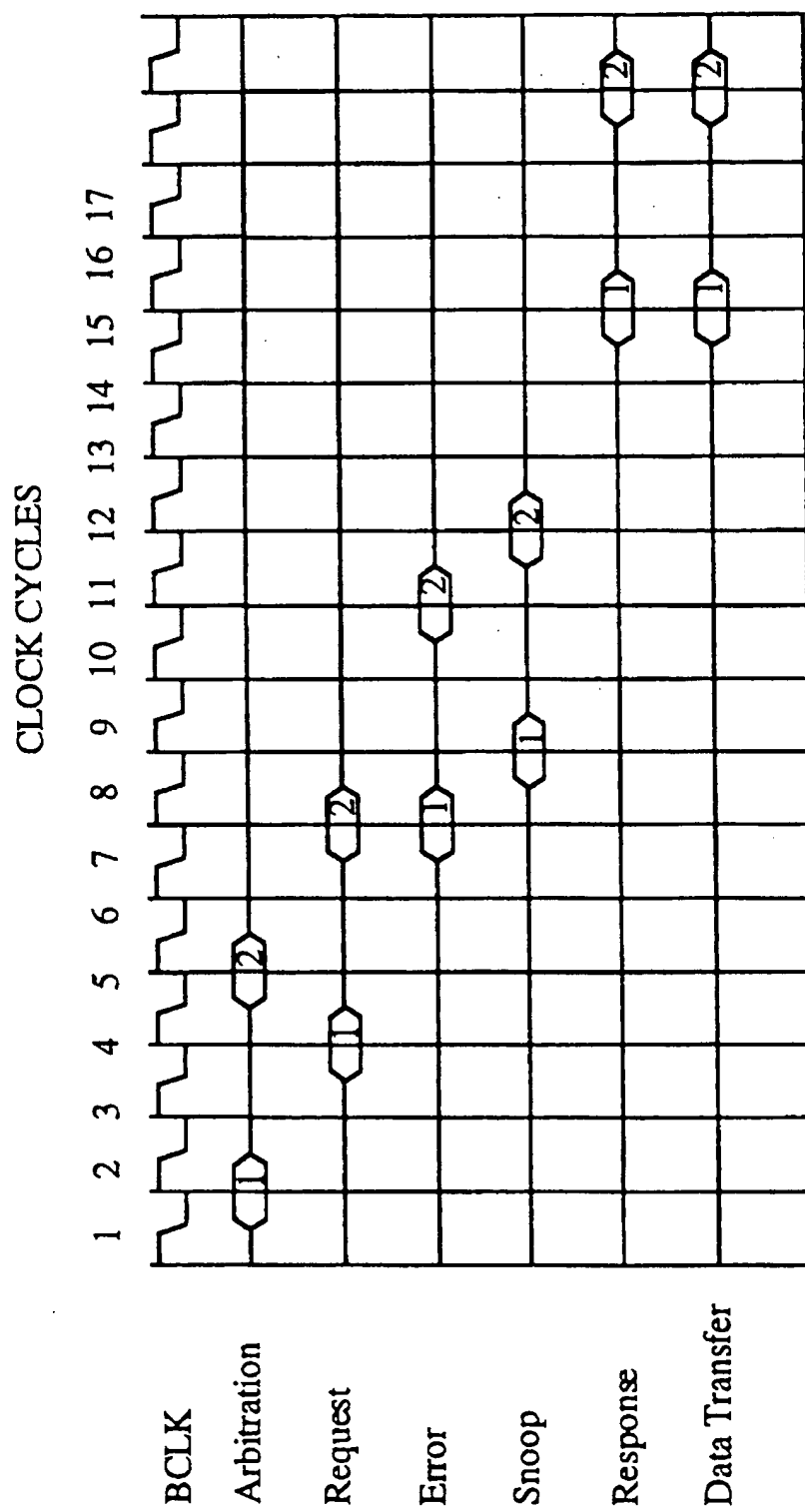
In accordance with the preferred embodiment of the present invention, a bus interface unit of a microprocessor is provided with a Micro Request Sequencer (EBMRS) disposed between a bus scheduling queue (EBBQ) and external bus control logic (EBCTL). Under normal bus request traffic, the EBMRS is effectively transparent and allows normal communication between the EBCTL and the EBBQ. However, for misaligned bus transactions, which comprise memory accesses that cross a bus width boundary, the EBMRS intercepts such transactions for special sequencing, while blocking any further requests from the EBBQ. The EBMRS separates each misaligned bus transaction request into at least first and second split transaction requests, with each split request forming a memory access that does not cross a data bus width boundary of the external bus. It then issues the first split request to the EBCTL for processing on the external bus. External bus agents involved with processing of the split requests then return first response information regarding the completion of the first split request. If the first response information indicates that the first split request will complete without being deferred or retried, the EBMRS issues the second split request to the EBCTL for processing on the external bus. Upon the receipt of second response information for the second split request indicating that the second split request is guaranteed to complete without being deferred or retried, the EBMRS then issues any further transaction requests received from the EBBQ without jeopardizing the order dependency of the split requests or subsequent bus transaction requests buffered in the EBBQ.

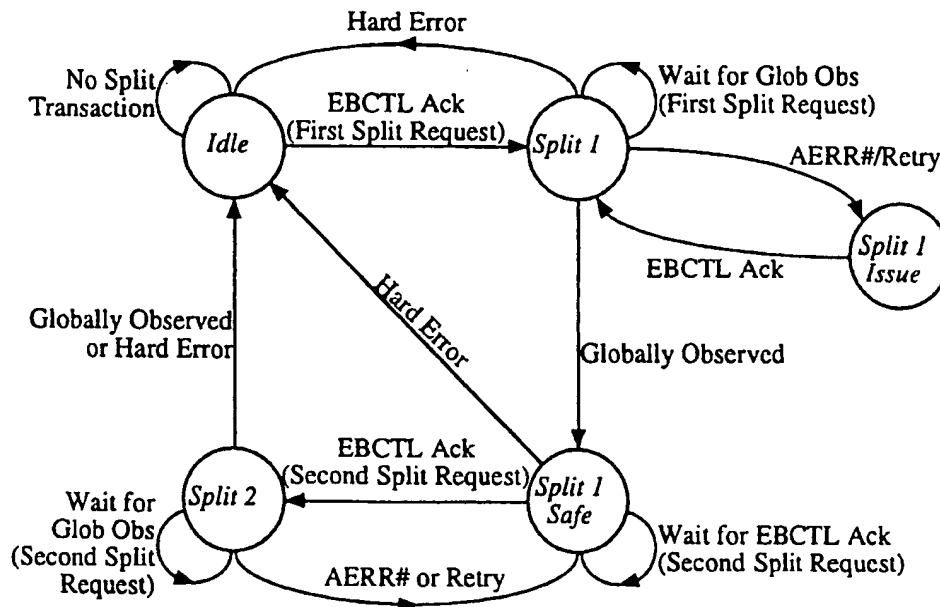
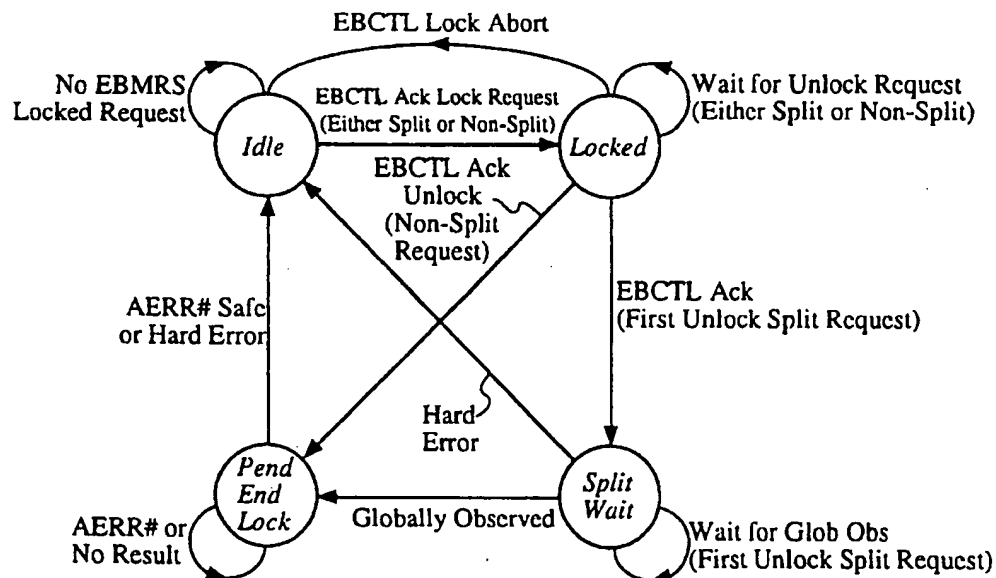
20 Claims, 5 Drawing Sheets



*Figure 1*

*Figure 2*

*Figure 3*

**Figure 4****Figure 5**

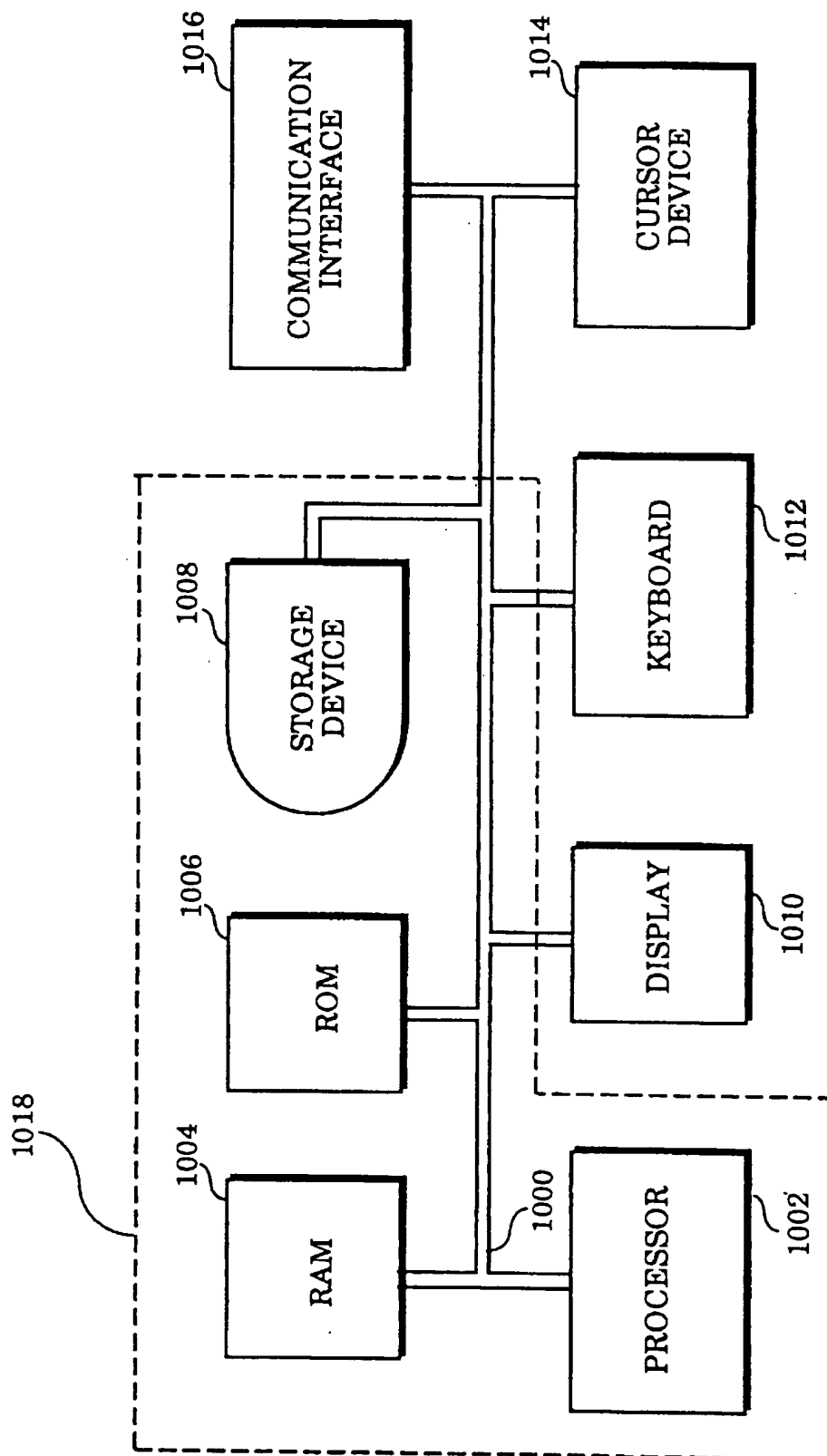


Figure 6

**METHOD AND APPARATUS FOR
SEQUENCING MISALIGNED EXTERNAL
BUS TRANSACTIONS IN WHICH THE
ORDER OF COMPLETION OF
CORRESPONDING SPLIT TRANSACTION
REQUESTS IS GUARANTEED**

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates to data communications in a computer system, and more specifically, to sequencing misaligned memory accesses forming misaligned bus transactions between a microprocessor and an external bus.

2. Art Background

In several conventional bus architectures, data transactions between the microprocessor and an external bus are permitted to start at any memory address with no alignment restrictions. This may result in requests issued to the external bus for memory accesses being split across at least two data bus width boundaries (i.e. each data bus being 8 bytes wide). Since the bus protocols for such microprocessors define a cache line data chunk return ordering which is not sequential with respect to the memory addresses (i.e., 8 bytes long), these processors must be able to sequence a misaligned memory access as two separate bus transactions in order to be compatible with the protocol of the external bus.

Conventional bus architectures have implemented sequencers for handling misaligned bus transactions. These sequencers take the original external bus request for the misaligned transaction issued by the microprocessor and split them into multiple requests, each of which represent a single transaction on the external bus that is within its data bus boundary width. Nonetheless, when an error occurs in processing of one of the split requests, the entire, original bus request must be canceled since no mechanism exists in such architectures for maintaining the ordering of the completion of the split requests so that a failed split request can be retried and completed in the order of its issuance.

This problem becomes even more significant for bus architectures and protocols in which the bus requests are to be issued and completed in-order on the external bus. This is because a misaligned bus transaction which must be canceled and reissued due to an error in processing of one of the split requests will cause a delay in the completion of other pending bus transactions that were issued subsequent to the canceled transaction, thereby resulting in a substantial performance penalty to the microprocessor.

Accordingly, it is an object of the present invention to provide an external bus micro-request sequencer for sequencing misaligned bus transactions comprising separate split requests in which the order of completion of the corresponding split requests is guaranteed.

It is another object of the present invention to provide a method and apparatus for sequencing misaligned bus transactions in a pipelined bus protocol wherein cancellation of the entire transaction due to an error in its corresponding split requests is avoided by preventing subsequent split and non-split requests from being serviced until the erred split request is guaranteed to complete.

SUMMARY OF THE INVENTION

The present invention provides a method and apparatus for sequencing the split requests of a misaligned external bus transaction in which the order of completion of the corre-

sponding split requests is guaranteed. In accordance with the preferred embodiment of the present invention, a bus interface unit is provided having a Micro Request Sequencer (EBMRS) disposed between a bus scheduling queue (EBBQ) and external bus control logic (EBCTL). Under normal bus request traffic (cache lines and aligned partial requests), the EBMRS is effectively transparent and allows normal communication between the EBCTL and the EBBQ.

However, for misaligned bus transactions, which comprise memory accesses that cross a bus width boundary, the EBMRS intercepts such transactions for special sequencing, while blocking any further requests from the EBBQ to the EBCTL for service. The original misaligned bus transaction is separated by the EBMRS into two "split" requests, which are then sequenced onto the external bus in order. During the sequencing, the EBMRS acts as a liaison between the EBCTL and the EBBQ so that all communication between them appears as if there were no split transactions in the system. Accordingly, the EBBQ submits single bus transactions as they arrive from the memory system, while the EBCTL sees only single directly mappable (split and non-split) transaction requests being issued from the EBMRS.

Split sequencing begins when the EBCTL accepts the first of two split requests issued from the EBMRS. The EBCTL treats the split request as a normal transaction, but the EBMRS waits to issue the second split request until the first split request is guaranteed to complete without being deferred or retried. After the request has been issued onto the bus by the EBCTL, the bus agents coupled to the external bus report back response information indicating whether the outstanding split request will be completed immediately, will be deferred without being retried due to a resource constraint or will simply have to be retried at a later time because of an error condition. This information is sampled by the EBCTL during the snoop phase of the transaction, and a global observation signal is generated and transmitted to the EBMRS when the response information indicates that the split request is guaranteed to complete in-order with respect to other previously issued bus transaction requests.

Upon receiving the global observation signal from the EBCTL, the EBMRS issues the second split request to the EBCTL for processing on the external bus since it is now known that completion of the first split request will precede completion of the second split request. When global observation is reported to the EBMRS for the second split request, the EBMRS then accepts the next external bus transaction from the EBBQ and issues that request to the EBCTL in the normal fashion assuming it is not misaligned. In this manner, the reporting of global observation to the EBMRS is utilized to determine when immediate completion of both the first and second split requests is guaranteed so that issuance of the second split request or a subsequent transaction, respectively, can be performed without jeopardizing the order dependency of the transactions.

In addition, the present invention also permits the sequencing of split requests for a misaligned bus transaction within a sequence of locked bus transactions. In the processing of special transactions on the external bus, such as read-modify-write transactions, the bus must be locked (i.e. exclusive ownership retained by the microprocessor) in order to permit strict sequencing of the locked bus transactions. To accomplish this, the present invention modifies the split sequencing described above so that upon receipt of a misaligned bus lock transaction or a misaligned bus unlock transaction, the EBMRS is able to perform split sequencing of the transaction while instructing the EBCTL when to lock the bus and when to unlock the bus so as to maintain system compatibility.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is generalized block diagram of one embodiment of the microprocessor in which the present invention is utilized.

FIG. 2 is a block diagram of the external bus logic and its interconnections to the external bus and various functional units of the microprocessor shown in FIG. 1.

FIG. 3 is a timing diagram in which the transaction phases for two external bus transactions having data transfers is depicted.

FIG. 4 is a state diagram for split transaction sequencing performed by the finite state machine of the micro-request sequencer.

FIG. 5 is a state diagram for split and non-split locked transaction sequencing performed by the finite state machine of the micro-request sequencer.

FIG. 6 is a block diagram of one embodiment of a computer system in which a microprocessor utilizing the present invention may be implemented.

DETAILED DESCRIPTION OF THE INVENTION

The present invention provides an external bus micro-request sequencer for sequencing the split requests of a misaligned external bus transaction in which the order of completion of the corresponding split requests is guaranteed. For purposes of explanation, specific embodiments are set forth in detail to provide a thorough understanding of the present invention. However, it will be apparent to one skilled in the art that the present invention may be practiced with other embodiments and without all the specific details set forth. In other instances, well known architectural elements, devices, circuits, process steps and the like are not set forth in detail in order to avoid unnecessarily obscuring the present invention.

I. Microprocessor Block Diagram

FIG. 1 is a generalized block diagram of one embodiment of a pipelined, speculative, out-of-order processor in which the external bus micro-request sequencer (EBMRS) of the present invention is preferably utilized. This particular embodiment includes a variety of functional units grouped together in clusters forming a bus cluster, an instruction fetch cluster, an issue cluster, an out-of-order cluster, an execution cluster and a memory cluster. More specifically, the functional units of this microprocessor comprise an instruction fetch unit (IFU), a branch target buffer (BTB), and instruction decoder (ID), a microinstruction sequencer (MS), an allocator (ALLOC), a register alias table (RAT), a reservation station (RS), a number of execution units (IEU, FEU, MIU), a reorder buffer (ROB) and a real register file (RRF).

Each instruction is decoded into one or more microinstructions or micro-operations (uops) by the ID with the help of the MS. The MS provides at least one microcode sequence in response to a corresponding microinstruction pointer (uip) that points to the first instruction in the sequence. The MS also implements special microcode sequences for handling events including exceptions and assists, and in one embodiment, emulates the handling routines used by the Intel microprocessor architecture.

The ID transfers the stream of uops to the RAT and the ALLOC. In one embodiment, the ID issues up to three in-order uops during each clock cycle of the processor. The

ALLOC assigns each incoming uop a location in the ROB in the same order as it was received from the ID, thereby mapping a logical destination address (LDST) of each uop to a corresponding physical destination address (PDST) of the ROB. The ALLOC maintains an allocation pointer pointing to the next ROB entry to be allocated. The ALLOC also receives a retirement pointer from the RCC, indicating which uops stored in the ROB are to be committed to architectural state in the RRF. Based upon the received retirement pointer, the ALLOC deallocates retired PDST entries of the ROB to make them available for reallocation.

The RAT maintains the mapping between LDST's and PDST's. To account for retirement, the RAT stores a real register file valid bit that indicates whether the value indicated by the LDST is to be found at the PDST entry of the ROB or in the RRF after retirement. Based upon this mapping, the RAT also associates every logical source address to a corresponding PDST entry of the ROB or the RRF (the source operand of one instruction generally must have been the destination of a previous instruction).

Each incoming uop is also assigned and written into an entry in the RS by the ALLOC. The RS assembles the uops awaiting execution by an appropriate execution unit (EU). When all the source operands of a uop are available and the appropriate EU (specified by an opcode) is ready, the uop is dispatched from the RS to the EU for execution. The EU then writes back result data and any appropriate flags via a writeback bus into the ROB at the entry indicated by the PDST of the dispatched uop. The EU also writes back into the PDST entry of the ROB event information which indicates whether an event has occurred, and if so, the type and specific nature of the event.

The ROB is implemented as a circular buffer that stores the speculatively executed results of micro instructions written back from the EU's. Once execution has completed and the result data of the uops are determined to be no longer speculative, the uops and their results are committed to architectural state in a process referred to as retirement. Uops stored in the ROB are retired in original program order into the RRF according to a retirement pointer. The retirement pointer is maintained in a retirement control circuit and points to all uops for which the processor has determined that the predicted instruction flow is correct. However, for uops which have caused an event, such uops will not be retired from the ROB when a microcode handler must be invoked in order to handle the event.

II. The External Bus Logic (EBL)

As shown in FIG. 2, the external bus logic (EBL) provides the physical and logical interface between the microprocessor (or a plurality of such microprocessors) and the external bus so as to enable external bus transactions requested by the microprocessor(s) to take place. The EBL also acts as a control interface between a level one data cache unit (DCU) and both the external bus and a level two (external) data cache unit (L2) via a backside bus logic unit (BBL). When a request for a cache access to the DCU fails to produce the required data, the EBL redirects the request to the L2 external cache via the BBL, and if another cache miss occurs, it then redirects the request to memory via the external bus.

To clarify terminology, a "transaction" is the set of bus activity that is related to a single bus memory access request. A transaction begins with bus arbitration, and the assertion of a signal ADS# along with a transaction address. Trans-

actions are issued to transfer data to inquire about or change cache states and/or to provide the system with information. A transaction contains up to six phases, with each phase using a specific set of signals to communicate particular information. However, not all transactions contain all phases, and some phases can be overlapped. The six phases of the bus protocol according to the preferred embodiment of the present invention are:

- Arbitration
- Request
- Error
- Snoop
- Response
- Data

With reference to FIG. 3, the bus transaction phases for two transactions having data transfers is depicted. When a requesting agent does not own the bus, a transaction begins with an arbitration phase in which the requesting agent becomes the bus owner. After the requesting agent becomes the bus owner, the transaction enters the request phase wherein the bus owner drives request and address information on the bus. The request state is two clocks long. In the first clock, the signal ADS# is driven along with the transaction address and sufficient information to begin snooping a memory access. In the second clock, more information such as byte enable information, deferred ID information and information concerning the length of the transaction are also driven.

Every transaction's third phase is an error phase, which occurs three clocks after the request phase begins. The error phase indicates any parity errors triggered by the request. Every transaction that is not canceled because an error was indicated in the error phase has a snoop phase four or more clocks after the request phase. The snoop results reported back to the EBCTL, EBMRS and EBBQ indicate if the address driven for a transaction references a valid or modified (dirty) cache line in any bus agent's cache in addition to whether a transaction is likely to be completed in-order with respect to previously issued transactions, be canceled and retried at a later time, or be deferred for possible out-of-order completion (when coupled to a processor that allows out-of-order completion of transaction requests on the bus).

Each transaction further has a response phase when the transaction is not canceled due to an error indicated in the error phase. During the response phase, the external bus agents report to the EBCTL information indicating whether the transaction has succeeded or failed (due to a hard error), whether the transaction is guaranteed to complete in order, whether the transaction's completion must be temporarily deferred, whether the transaction will be retried and whether the transaction contains a data phase. If the transaction does not have a data phase, that transaction is complete after the response phase. However, if the requesting agent has write data to transfer, or has requested read data, the transaction enters the data phase which may extend beyond the response phase.

Referring again to FIG. 2, the external bus logic comprises four major functional units: the external bus request queue (EBBQ), the microrequest sequencer (EBMRS), the external bus protocol control logic (EBCTL) and the external bus pad logic (EBPADS). The EBBQ is responsible for processing transaction requests from the DCU and the instruction fetch unit (IFU). It delivers to the BBL those requests that require L2 service and forwards those that the L2 cannot process successfully (i.e., L2 cache miss) to the external bus. Hence, the EBBQ can be viewed as the

transaction scheduler for the EBL/BBL units. The EBMRS provides support for misaligned data as required by backward compatibility. It examines requests from the EBBQ to determine if a request crosses a bus width boundary. If a bus width crossing is detected, the EBMRS splits the single request from the EBBQ into two separate "split requests" and sequences them to the EBCTL. If the request is not misaligned, only a single request is presented to the EBCTL. The EBMRS is also responsible for generating the correct byte enables, cacheability attributes and deferred ID for external bus requests.

The EBCTL interfaces the microprocessor(s) to the external world and implements the external bus protocol. It is responsible for forwarding EBBQ/EBMRS requests to the external bus, informing the EBMRS and EBBQ of external request activity, transferring requested data between the BBL/DCU/L2 and the external bus, and tracking the status of all outstanding requests on the bus. Finally, the EBPADS contains the external bus I/O latches, the IEEE 1149.1 boundary scan cells and functional redundancy check logic.

With reference to FIGS. 2 and 3, an overview of the operation of the EBL units in processing transaction requests will be described below. Transaction requests are sent to the EBL by the IFU and DCU. Requests are processed and acknowledged by the EBBQ. The EBBQ forwards request to the BBL and the external bus based on the cacheability of the request type and whether or not the requested cache line is contained in the L2 cache. Requests which are non cacheable or have resulted in an L2 miss are forwarded to the EBMRS which handles misaligned bus accesses by splitting them into two aligned requests. The EBMRS forwards all requests including split requests to the EBCTL which implements the external bus protocol and sends the request to the external bus. The EBCTL tracks all bus requests and sends information regarding them to the other blocks of the EBL. All information flowing between the EBL and the external bus is latched by the EBPADS. When the response time of a transaction occurs, EBCTL takes care of sending the data to or from the processor core and informs the EBBQ when the transaction can be deallocated.

In performing read transactions, the EBL first sends a read request to the BBL for L2 processing. The BBL notifies the EBL whether the request resulted in an L2 hit or miss. If the request resulted in an L2 hit, the data is immediately returned to the DCU/IFU. In the event of an L2 miss, EBL sends an internal request for the transaction. If the microprocessor does not have control of the address (request) bus, EBL will arbitrate for the bus. If the microprocessor already has control of the bus, EBL will issue the request to the bus without entering arbitration. The EBL may have to assert the internal request for one or more cycles until the microprocessor actually arbitrates for and/or sends the request to the bus. If a request error is reported on the bus at this time, then the request is canceled and retried at a later time. Otherwise, the read transaction proceeds to the snoop phase where the owner of the data transfer is determined and snoop results, such as a cache hit signal and a defer signal, are reported to the other units. If necessary, the microprocessor bus protocol allows for the snoop phase to be stalled by any microprocessor bus agent.

Once the data transfer owner has been determined, the response agent (typically the addressed bus agent) drives response information for the request to the EBCTL indicating whether the transaction will be completed in order with respect to previously issued transaction requests, have its completion deferred without retrying the transaction or have its completion deferred until the transaction is retried. If a

"normal" response is received, one or four —64 bit data chunks are returned on the bus and forwarded to the BBL/DCU/IFU. If a "defer" response is given, the response agent must later complete the transaction as a deferred reply transaction which may cause it to complete out-of-order. If a "retry" response is received another internal request for the transaction is transmitted by the EBL, and the transaction is retried on the external bus. If an "error" response is received, the transaction does not proceed beyond the response phase as it is typically canceled. Once all data chunks have been received, a transaction complete signal is sent to the BBL/DCU/IFU to end the transaction.

For write transactions to the external bus, the operation of the EBL is similar to that for read transactions except that data is retrieved from the BBL/DCU rather than sent to it. Also write transactions have a split response phase where the first part of the response initiates the data transfer and the second part of the response completes the transaction. Notice that the first part of the response can happen concurrently with the request error phase. However, transaction cannot complete until the microprocessor transfers all the write data onto the external bus and the final response is given by the response agent.

III. Global Observation of Transaction Completion

In highly pipelined bus architectures where bus transactions are allowed to complete out of order due to resource constraints or error conditions, it is desirable to provide a mechanism for informing bus agents when a particular transaction will be guaranteed in order completion so as to maintain memory consistency and avoid a shortage of resources. In the preferred embodiment, this is accomplished by defining in the microprocessor's external bus protocol a synchronizing (or memory ordering) event called "global observation." This event is an agreed upon point at which an agent responding to a transaction reports to other bus agents participating in the transaction and to the EBCTL that the transaction is guaranteed to complete in-order with respect to other previously issued transaction requests (i.e., will not be retried or deferred), thereby allowing the requesting agent to proceed with other order dependent transactions.

Upon issuance of a request for a transaction on the bus, all agents which may participate in the transaction must inform all other bus agents (in order) as to whether or not the specific actions requested by a transaction will be completed in-order. During the snoop phase of the transaction, ownership of the data transfer is determined and completion information for the outstanding transaction is collected from the external bus agents and reported to the EBCTL. The completion information sent to the EBCTL comprises a cache hit signal indicating whether the data has been found to reside with a particular bus agent and a defer signal indicating whether the transaction may be deferred due to a resource constraint or retried due to an error condition.

The EBCTL samples these signals to determine whether the transaction can be guaranteed to complete (in order) at this time. If the defer signal is not asserted, this indicates that the transaction request can be satisfied immediately, and hence, that the transaction's completion is guaranteed. If the defer signal is asserted, then one of two cases may result: If the cache hit signal is also asserted, then completion of the transaction can still be guaranteed at this time since a bus agent having the desired data has been identified. However, if the cache hit signal is not asserted, then either the transaction's completion is deferred until the necessary resources are available, or the entire transaction is retried by

reissuance of the request from the EBBQ depending upon the response information generated in the response phase.

If the transaction can be guaranteed to complete at this time, then the EBCTL transmits to each of the EBBQ and the EBMRS a global observation signal indicating that:

- 1) the cache state transition requested by the transaction has taken place.
- 2) the agent granting global observation has acquired or relinquished snoop ownership of the transaction.
- 3) data will be transferred (if applicable).
- 4) a normal completion response (barring a hard error) will be sent for the transaction, thereby allowing it to complete in-order.

In this manner, both the EBBQ and the EBMRS can continue with other order dependent transactions upon receiving global observation for the previously issued, outstanding transaction. If, however, completion of a particular transaction cannot be guaranteed during this phase, then the EBBQ and EBMRS must wait until they receive either a retry response (meaning that the transaction will be canceled and later re-issued by the bus queue) or a deferred response (meaning that the transaction will complete sometime later and possibly out-of-order) in order to proceed with further transactions.

IV. The External Bus Request Queue (EBBQ)

The EBBQ is a centralized queuing structure that collects bus access requests from the IFU and DCU, then schedules those requests for in-order issuance to the L2 and external bus. It tracks the status of all requests from the L1 cache from the moment they are accepted until the final completion handshake back to the caches. This tracking allows the bus queue to correctly sequence requests and handle external snoops to requests that are in progress.

When a request requires external bus service, the EBBQ will issue a request to the EBCTL via the EBMRS. Together they implement the mechanisms for generating, sequencing and tracking requests on the external bus. To send a request, the EBBQ asserts a request indication and drives the address and request attributes. In request terminology, driving of the address pins is broken into two different cycles; the cycle during which ADS# is active is referred to as a first ADS# cycle, while the cycle after ADS# was active is referred to as a second ADS# cycle.

Within the EBBQ, the order of selection of pending requests for issuance onto the external bus is made by a pointer rotating between the different EBBQ entries, with the EBBQ having four entries in the preferred embodiment. When the pointer points to a valid queue entry, it is presented to the EBMRS/EBCTL as described above. The pointer stays there, and hence the request stays presented, until the acknowledgment is received. When rotating, the pointer advances one queue entry per clock cycle.

Once a request is presented to the external bus by the EBBQ, it will not be changed until an acknowledgment is received from the external bus. This will occur after the second ADS# cycle has been driven onto the bus for normal requests, and after the second ADS# cycle of the last of two split requests has been driven onto the bus in the case of a partial operation which resulted in a split transaction.

In the completion phase, the external bus reports the completion of a request on the external bus. There are two distinct completion signals generated by the EBCTL because two separate requests may need to be completed in

the same cycle. The complete signals are accompanied by a bus queue index, as well as the buffer id which was passed to the EBCTL when the request was issued. In addition to those requests which reside in the EBBQ, the EBBQ may receive a complete indication for a buffer which is being used for an implicit writeback, and as such may not have a corresponding entry in a queue.

The EBBQ has a private Completion path back to the DCU and shares a common completion path back to the IFU. In both cases, the EBBQ will signal Completion to the L1 cache when it has received Completion from the external bus, and has already forwarded all of the chunks received from the external bus, to the corresponding L1 cache. Each completion signal is accompanied by an error indication, which is asserted in the case that a Hard Error or Double Parity Error occurred while issuing the request onto the bus. The receiving L1 cache in these instances is required to take the appropriate error recovery actions.

V. The External Bus Micro Request Sequencer (EBMRS)

The EBMRS is the interface between the EBBQ and the EBCTL for handling the sequencing of split and lock transactions to the external bus. All core requests to the EBBQ that cannot be satisfied by the L2 via the BBL are presented to the EBMRS for service on the external bus. The EBMRS examines these requests (i.e., a "split field" of the request) to determine if it is a split transaction (i.e., partial reads and writes that cross 64 bit cache line chunk boundaries) and makes a request to the EBCTL for service.

If no split transaction is detected, the EBMRS translates the request from internal microprocessor format to the format of the external bus before passing it to the EBCTL. If a split transaction is detected the EBMRS dynamically translates the EBBQ's internal request and sequences this request into multiple "split requests" on the external bus such that each split request appears as a single external bus transaction and is contained entirely within a cache line. This includes a calculation of the appropriate byte enables for each split request and tracking the entire life of the split request up to a global observation phase where it is then determined whether or not the split request will be completed in order with respect to its issuance from the EBBQ. Some EBCTL signals associated with the requests' life on the bus are directly sent to the EBBQ (e.g., transaction completion signals), whereas other information is routed through the EBMRS (e.g., global observation, acknowledgment, etc.).

The EBMRS decodes the EBBQ's core request to determine the external bus transaction type. This information is used to drive the REQ# pins which encode the request type and accompany every ADS# on the external bus. The EBBQ presents requests to the EBMRS via a request signal, an encoded request type, the transaction address and length, and any request attributes. The request type and attributes allows the EBMRS to completely specify the entire two cycle external bus request encoding, which appear on the REQ# and A3# pins during and the clock after an ADS# for microprocessor issued transactions.

The EBBQ makes a request to the EBMRS for external bus service when it has determined that the BBL cannot service the request at hand. Certain requests are first presented to the BBL to see if they can be satisfied there, whereas others are directly presented to the EBMRS for issuance onto the external bus. After the EBMRS has formed

all the request attributes, it makes a request to the EBCTL for an external bus transaction. When this request is made to EBCTL, the corresponding address and request pin information is checked for parity errors and then sent to the EBPADS with an ADS# for driving of the request on the external bus. A special signal is sent to EBCTL if the current request is a bus lock, and will require assertion of the LOCK# pin per the bus protocol. The EBMRS also informs EBCTL whether or not each request is a cache line or a partial request, and whether or not EBCTL should send a complete signal to the EBBQ upon completion of this transaction.

There are several types of requests that the core makes to the EBBQ, and several types that the EBBQ presents to the EBMRS, including Read, Read for Ownership, Locked Read for Ownership, Invalid to Modified, Write Back, Write Combining Chunk Eviction, Write Combining Line Eviction, Partial Read, Partial Write, IO Read, IO Write, Locked Partial Read, Split Locked Partial Read, Locked Partial Write, Special Cycle, Interrupt Acknowledge, and Branch Trace Message transactions. However, these external bus transactions can be more broadly classified into reads and writes (line or partial or I/O), code reads, invalidates, Interrupt Acknowledge, Branch Trace Message, and various "Special Bus Cycles". As mentioned above, each of these external bus transactions consist of an ADS# accompanied by two consecutive clocks of request information on the address bus A# and the request pins REQ#.

VI. Split Transaction Sequencing

From the perspective of the external bus, a misaligned bus transaction is a transaction whose address/length combination crosses an aligned chunk boundary and the bus ordering protocol (i.e., the processor line transfer ordering) does not match the cache line transfer ordering of the memory subsystem. The DCU can issue partial read/write transactions (locked or not) with an address and length combination such that the single DCU transaction will require bytes from two separate 8 byte chunks which may be non-contiguous with respect to the bus ordering protocol. Since the data bus is 8 bytes wide, the external bus must sequence such transactions as two separate requests, each of which returns one 8 byte chunk of data with the appropriate byte enables set for each. It is the EBMRS' responsibility for sequencing such transactions to the EBCTL in the proper fashion. Because a single DCU transaction can result in two bus transactions, special care must be taken to closely track the progress of each split request forming an entire split transaction. The EBMRS receives global observation, error and defer/retry/hard failure response information for each split request from EBCTL.

The EBBQ determines whether a given transaction is a split transaction or not by checking to see if the number of bytes requested by the DCU crosses a chunk boundary when added to the transaction address. This information is conveyed to the EBMRS by means of the EBBQ setting a bit in the split field of the request in addition to asserting the EBBQ's request line. Alternatively, the EBMRS could itself determine whether a split transaction is required by checking the number of bytes which cross a chunk boundary so as to relieve the EBBQ of this responsibility. The EBMRS is then responsible for splitting the transaction into two separate "split" requests (each with an 8 byte aligned address) and for setting the appropriate address and byte enable information for each. Additionally, the EBMRS receives global observation information from the EBCTL, which tells the

EBMRS when it is safe to initiate the second split request, while asserted error information is also communicated to the EBMRS from the EBCTL.

Although in the preferred embodiment it is assumed that an internal (or microprocessor) request comprises a request for an 32 byte data cache line, and hence, that a split transaction will be split into only two split requests, it is envisioned that the internal transaction request may actually cross more than one cache line boundary of the memory subsystem (depending upon the units of transfer on the bus and the data sizes supported by the processor) and therefore require separation into more than two split requests. Accordingly, the present invention may be adapted to encompass this situation by providing logic in either the EBBQ or the EBMRS which compares the number of bytes in the transaction request to the standard data cache line width of the memory subsystem and determines the number of cache line boundaries that the transaction request will cross when added to the transaction address.

Split sequencing begins when EBCTL accepts the first of two split requests, and ends when the second split request has passed its global observation window during the snoop phase of the transaction. The EBMRS keeps track of which split request is in progress, and reports global observation on a split request to the EBBQ only when the second split request has passed its global observation window. As the EBBQ does not track the progress of each individual split request, the EBMRS must inform EBCTL whether or not to send a complete signal to the EBBQ with each split request. This is because a complete signal for all transactions is sent directly to the EBBQ from EBCTL. (Unlike, for example, global observation, and acknowledge signals, which are routed through the EBMRS.) The Send Complete signal accompanies all normal transactions made by the EBMRS to EBCTL, but for split transactions, it is deasserted on the first split request of the split transaction so the transaction is not de-allocated before the second split request is complete.

The EBMRS contains special sequencing logic to track the progress of each split request as is shown in the state diagram of FIG. 4. In the Idle state, information regarding all transactions is passed directly back to the EBBQ from the EBCTL. The split sequencing begins when the first split request of a split transaction is accepted by the EBCTL (Split 1 state). The EBMRS remains in this state until the EBCTL samples the appropriate signals and reports global observation for this first split request to the EBMRS. If an address request error or retry response is received by the EBMRS, it will retry the first split request once more (Split 1 Issue state). A receipt of a hard error while in any active split sequencing state will result in an immediate return to the Idle state and canceling of the entire split transaction.

Upon receipt of global observation for the first split request, the EBMRS enters the Split 1 Safe state and then issues the second split request to the EBCTL. When the second split request is acknowledged by the EBCTL, the Split 2 state is entered. If an address request error or retry response is received for the second split request, the EBMRS will return to the Split 1 Safe state and reissue the second split request. The EBMRS then returns to the Idle state when global observation is received for the second split request. In this manner, the reporting of global observation from the EBCTL to the EBMRS is utilized to determine when in-order completion of both the first and second split requests is guaranteed so that issuance of the second split request or a subsequent transaction, respectfully, can be performed without jeopardizing the order dependency of the transactions.

It is noted that if more than two split requests were required by a split transaction that crosses more than one data chunk boundary, the split sequencing shown in the state diagram of FIG. 4 would have to be extended to include (after the split 2 state) a split 2 safe state and a split 3 state, etc., with appropriate returns provided to the idle state and the split 2 safe state upon receiving a hard error or either an address request error or retry, respectively. With such an extension, global observation would still be reported for each preceding split request before the succeeding split request is issued to the EBCTL.

According to an alternate embodiment, it is not necessary to wait for global observation to be reported by the EBCTL before issuing each split request of a split transaction. In this embodiment, the EBMRS would keep a "scoreboard" of small registers to track the status of each split request since the ordering of the completion of the split requests cannot be guaranteed. Again, this mechanism could be extended to accommodate bus transactions which are broken into more than 2 split requests.

VII. Locked Transaction Sequencing

Locked bus transaction sequences (typically consisting of at least one partial read followed by the same number of partial writes) require special handling by the EBBQ because of the need for strict sequencing of the transactions to the external bus for system compatibility. To accomplish this, the EBBQ enters a mode in which it first drains itself of all previously existing requests, and then steps one by one through a sequence of locked bus transactions upon locking of the bus for exclusive use by the microprocessor. In order to indicate the start of a locked sequence to the EBL units, the first partial read transaction should be of the lock or split lock variety, whereas to indicate the end of the lock sequence, the last partial write transaction should be of the unlock or "split" unlock variety.

When a lock read request is presented to the EBBQ, all outstanding requests are completed, and the lock request is accepted by the EBBQ. The EBBQ continues accepting locked requests until an unlock write request is received and accepted, after which the queue is drained and the request acceptance policy is returned to normal. During lock processing, new DCU requests can be accepted in the normal, lock request acceptance and DCU acceptance states, while new IFU requests can only be accepted in the normal state.

Since the EBMRS contains intimate knowledge of split requests, it is efficient for the EBMRS to handle lock sequencing because the microprocessor bus lock protocol does not match exactly with the lock protocol emanating from the memory system. Therefore, the EBMRS is responsible for tracking all transactions that will be issued on the external bus with the LOCK# pin asserted. Lock and unlock requests are identified by the EBMRS by detecting certain bits of the EBBQ's request type, including DCU lock partial reads and unlock partial writes. In addition, the special DCU split lock read transaction is also supported. This is issued by the DCU when a given transaction crosses a cache line boundary, but remains within a chunk boundary. Read-modify-write sequences consist of lock partial read transactions followed by unlock write transactions from the DCU. This ensures that the bus does not hang due to the LOCK# pin being asserted indefinitely.

Legal DCU read-modify-write sequences could include a locked partial read followed by a locked partial write request. In this case, the lock variable may result in a

(chunk) split access on the bus. This means that there can be at most four consecutive ADS# cycles corresponding to two locked partial reads and two locked partial writes on the bus. In the absence of an error, the LOCK# pin will stay asserted from the time the first ADS# was asserted, until at least the response phase of the last locked transaction. It is noted that locked transactions cannot receive a deferred response (except where the transaction is the first transaction of a locked sequence, in which case, such a response will cause the entire lock sequence to be aborted).

With reference to the sequencing of locked bus transactions performed by the EBMRS, as shown in the state diagram of FIG. 5, the EBMRS normally resides in the idle state. If a lock request is received from the EBBQ, the EBMRS issues the request to the EBCTL and instructs it to lock the bus. If the lock transaction is of the split variety, the EBMRS handles it as a normal split transaction by separating it into two split requests and processing it according to the state diagram of FIG. 4. Nonetheless, the EBMRS instructs the EBCTL to lock the bus upon the issuance of the first split request. The EBCTL proceeds to lock the bus, and upon acknowledging the lock request (either split or non-split) to the EBMRS, the Locked state of the EBMRS's state machine is entered. The EBCTL continues to lock the bus until instructed to unlock it by EBMRS.

The EBMRS remains in the Locked state during processing of the subsequent locked bus requests until an unlock request (either split or non-split) is issued from the EBBQ or a lock abort signal (lock error or restart) is reported by the EBCTL. When an unlock transaction is received from the EBBQ, the EBMRS checks to see whether it is a split or non-split transaction. If it is a non-split transaction, the EBCTL is instructed to unlock the bus upon issuance of the request to the EBCTL. However, if it is a split transaction, the EBMRS separates it into two split requests and issues the first split request to the EBCTL without instructing the EBCTL to unlock the bus. Accordingly, the state machine of the EBMRS can enter one of two states once the unlock request is accepted by the EBCTL. If the unlock request is a non-split request, the EBMRS will enter the Pend End Lock state upon receiving acknowledgment from the EBCTL for the request. If the unlock request is the first part of a split unlock transaction, the Split Wait state is entered upon acknowledgment of the first split unlock request. Upon reporting of global observation of the first split unlock request from the EBCTL to the EBMRS, the EBMRS enters the Pend End Lock state where the second unlock split request is issued to the EBCTL. The EBMRS also instructs the EBCTL to unlock the bus upon issuance of the second split request, however, the EBCTL does not actually unlock the bus until the second unlock split has safely passed through its response phase without errors as indicated by the response information returned from the bus agents. Note that if the first split unlock request had resulted in a hard error, then the entire lock sequence would be aborted, returning the state machine of the EBMRS to the idle state.

The Pend End Lock state is the final state before the lock cycle is forgotten by the EBMRS. Once having entered the Pend End Lock state, the EBMRS remains in this state until the final unlock request (split or non-split) has safely passed through its request error phase (the earliest that the bus can be unlocked). Upon receipt of response information from the EBCTL regarding safe passage of the transaction through its request error phase (or a hard failure response) for the final unlock request, the EBMRS exits the lock cycle and returns to the Idle state. If a request error is received for the transaction (split or non-split) in the Pend End Lock state,

the transaction is retried on the bus with a reaffirmation to the EBCTL that the bus should be unlocked.

In an alternate embodiment, the EBMRS can issue to the EBCTL the second unlock split request for a split transaction while in the Split Wait state as soon as the first unlock split request has been globally observed. The transition to the Pend End Lock state is then made after the final split unlock request has been acknowledged by the EBCTL. This therefore makes the state transitions into the Pend End Lock state more symmetric and further enables the Pend End Lock state to perform a single function regardless of whether the transaction is split or not.

Note that blocking of EBBQ requests to the EBCTL by the EBMRS is related to split sequencing only. The EBMRS acts as a passive device for normal locked requests (only instructing the EBCTL when to lock and unlock the bus). Although split sequencing can take place within a locked sequence, and may have an effect on the lock sequence state, the locked state does not directly cause the EBMRS to affect the communication between the EBCTL and bus scheduling queue.

VIII. Computer System

As mentioned earlier, the present invention described above can be implemented in an out-of-order microprocessor or in more conventional pipelined and non-pipelined microprocessors. Regardless of the type of processor in which the invention is used, the processor selected may be added to a general purpose computer system as shown in FIG. 6.

Such a computer system comprises an address/data bus 1000 for communicating information, a central processor 1002 coupled with the bus for processing information and executing instructions, a random access memory 1004 coupled with the bus 1000 for storing information and instructions for the central processor 1002, and a read only memory 1006 coupled with the bus 1000 for storing static information and instructions for the processor 1002. Also available for interface with the computer system of the present invention is a data storage device 1008 such as a magnetic disk or optical disk drive, which may be communicatively coupled with the bus 1000, for storing data and instructions.

The display device 1010 utilized with the computer system of the present invention may be a liquid crystal device, cathode ray tube, or other display device suitable for creating graphic images and/or alphanumeric characters recognizable to the user. The computer system may also contain an alphanumeric input device 1012 including alphanumeric and function keys coupled to the bus 1000 for communicating information and command selections to the central processor 1002, and a cursor control device 1014 coupled to the bus 1000 for communicating user input information and command selections to the central processor 1002 based on a user's hand movement.

The computer system of FIG. 6 also contains an input/output device 1016 coupled to the bus 1000 for communicating information to and from the computer system. The communication device 1016 may be composed of a serial or parallel communication port or may be a communication modem. It is appreciated that such a communication device 1016 may provide an interface between the bus 1000 and the user interface devices (keyboard 1012, cursor 1014, display 1010) of the computer system. In this case, the user interface devices will reside within a terminal device which is coupled

15

to the communication device 1016 so that the processor 1002, the RAM 1004, the ROM 1006 and storage device 1008 may communicate with the terminal. The components 1002, 1004, 1006 and 1008 may be implemented on a single board or a computer chassis 1018, which is then coupled by a bus 1000 to the other components of the computer system.

It will be appreciated that various modifications and alterations might be made by those skilled in the art without departing from the spirit and scope of the present invention. Therefore, the invention should be measured in terms of the claims which follow.

We claim:

1. In a microprocessor having an associated bus interface unit with control logic for coupling the microprocessor to an external bus and processing external bus transaction requests on the external bus, a method is provided for sequencing misaligned bus transaction requests on the external bus, each of said misaligned bus transaction request forming a memory access crossing a data bus width boundary of the external bus, the method comprising the steps of:

said bus interface unit of said microprocessor separating each misaligned bus transaction request into at least first and second split transaction requests, with each split request forming a memory access that does not cross a data bus width boundary of the external bus;

issuing the first split request to the control logic of the bus interface unit for processing of the first split request on the external bus;

determining whether a global observation has been issued for said first split request;

in response to said global observation of said first split request issuing the second split request to the control logic of the bus interface unit for processing of the second split request on the external bus;

determining whether a global observation has been issued for said second split request; and

in response to said global observation of said second split request, completing processing of the second split request on the external bus in order with respect to issuance of the first split request.

2. The method of claim 1, wherein the bus interface unit further comprises a bus queue for buffering and issuing requests and a transaction request sequencer for sequencing the requests received from the bus queue to the control logic, and before the step of separating each misaligned bus transaction request into at least first and second split transaction requests, the method further comprises the steps of:

transmitting external bus transaction requests from a requesting unit of the microprocessor to the bus queue;

issuing the bus transaction requests from the bus queue to the transaction request sequencer; and

identifying bus transaction requests which cross a data bus width boundary of the external bus as misaligned bus transaction requests.

3. The method of claim 2, wherein the steps of determining whether the global observation of said first split request has been issued and issuing the second split request to the control logic of the bus interface unit in response to said global observation is performed by the steps of:

transmitting said global observation from bus agents coupled to the external bus to the control logic of the bus interface unit, the global observation indicating whether the first split transaction request will complete without being deferred or retried;

transmitting a first completion guarantee signal from the control logic to the sequencer when the first response

16

information indicates that the first split transaction request is guaranteed to complete without being deferred or retried; and

issuing the second split request from the sequencer to the control logic of the bus interface unit for processing of the second split request on the external bus when the first completion guarantee signal for the first split request is received by the sequencer from the control logic.

4. The method of claim 2, wherein the steps of determining whether the global observation of said second split request has been issued and completing processing of the second split request on the external bus in order with respect to issuance of the first split request in response to said global observation of said second split request is performed by the steps of:

transmitting said global observation from bus agents coupled to the external bus to the control logic of the bus interface unit, the global observation indicating whether the second split transaction request is guaranteed to complete without being deferred or retried;

transmitting a second completion guarantee signal from the control logic to the sequencer when the second response information indicates that the second split transaction request is guaranteed to complete without being deferred or retried; and

completing processing of the second split request on the external bus when the second completion guarantee signal for the second split request is received by the sequencer from the control logic.

5. The method of claim 2, wherein the step of identifying bus transaction requests which cross a data bus width boundary of the external bus as misaligned bus transaction requests is performed by one of the bus queue and the sequencer.

6. The method of claim 1, wherein the method further comprises the step of canceling the misaligned bus transaction request upon the occurrence of a hard error.

7. In a microprocessor having an associated bus interface unit with control logic for transferring data between the microprocessor and external bus agents via an external bus, a method is provided for sequencing misaligned bus transaction requests on the external bus, each said misaligned bus transaction request forming a memory access crossing a data bus width boundary of the external bus, the method comprising the steps of:

transmitting external bus transaction requests generated by the microprocessor to the bus interface unit;

determining whether the data to be transferred in each external bus transaction request will cross a data bus width boundary of the external bus;

said bus interface of said microprocessor separating each external bus transaction request denoted as a misaligned bus transaction request into at least first and second split transaction requests when the data to be transferred in the external bus transaction request will cross a data bus width boundary of the external bus, with each split request forming a memory access that does not cross a data bus width boundary of the external bus;

issuing the first split request of each misaligned bus transaction request to the control logic of the bus interface unit;

transmitting request and address information corresponding the first split request from the control logic to the bus agent specified by the address information to effect

a transfer of data between the microprocessor and the addressed bus agent;

transmitting a first global observation from the bus agents to the control logic of the bus interface unit;

issuing the second split request to the control logic of the bus interface unit when the first global observation transmitted from the bus agents is received by said control logic;

transmitting a second global observation from the bus agents to the control logic of the bus interface unit; and

completing processing of the second split request on the external bus in order with respect to issuance of the first split request when the second global observation is received by said control logic.

8. The method of claim 7, wherein for a misaligned bus transaction request comprising a bus lock transaction request of a sequence of locked transaction requests for which the external bus must be locked for exclusive use by the microprocessor in processing the sequence of locked transaction requests, the method further comprises the step of locking the external bus for exclusive use by the microprocessor upon issuance of the first split request of the misaligned bus lock transaction request to the control logic.

9. The method of claim 7, wherein for a misaligned bus transaction request comprising a bus unlock transaction request of a sequence of locked transaction requests for which the external bus must be locked for exclusive use by the microprocessor in processing of the sequence of locked transaction requests, the method further comprises the step of unlocking the external bus to enable use of the external bus by other bus agents when the second global observation for the second split request is received from the external bus agents.

10. The method of claim 7, wherein the method further comprises the step of canceling the misaligned bus transaction request upon the occurrence of a hard error.

11. In an out-of-order microprocessor comprising an instruction fetch unit, a data cache unit and an associated bus interface unit for transferring data between one of the instruction fetch unit and the data cache unit and external bus agents via an external bus, a method is provided for sequencing misaligned bus transaction requests on the external bus, each misaligned bus transaction request forming a memory access crossing a data bus width boundary of the external bus, the bus interface unit having control logic capable of processing external bus transaction requests in order of receipt from the instruction fetch unit and the data cache unit, the method comprising the steps of:

A) transmitting external bus transaction requests from one of the instruction fetch unit and the data cache unit to a bus queue of the bus interface unit for buffering of the bus transaction requests;

B) issuing the bus transaction requests from the bus queue to a transaction request sequencer of the bus interface unit;

C) identifying bus transaction requests which cross a data bus width boundary of the external bus as misaligned bus transaction requests;

D) separating each misaligned bus transaction request into at least first and second split transaction requests, with each split request forming a memory access that does not cross a data bus width boundary of the external bus;

E) issuing the first split request of the misaligned bus transaction request from the sequencer to the control logic of the bus interface unit for processing of the first split request on the external bus;

F) transmitting first response information from the bus agents coupled to the external bus to the control logic of the bus interface unit, the first response information indicating at least whether the first split transaction request will complete without being deferred or retried;

G) transmitting a first completion guarantee signal from the control logic to the sequencer when the first response information indicates that the first split transaction request is guaranteed to complete without being deferred or retried;

H) issuing the second split request from the sequencer to the control logic of the bus interface unit for processing of the second split request on the external bus when a first completion guarantee signal for the first split request is received by the sequencer from the control logic;

I) transmitting second response information from the bus agents to the control logic of the bus interface unit, the second response information indicating at least whether the second split transaction request is guaranteed to complete without being deferred or retried;

J) transmitting a second completion guarantee signal from the control logic to the sequencer when the second response information indicates that the second split transaction request is guaranteed to complete without being deferred or retried; and

K) returning to step B) when the second completion guarantee signal for the second split request is received by the sequencer from the bus control logic.

12. The method of claim 11, wherein for a misaligned bus transaction request comprising a bus lock transaction request of a sequence of locked transaction requests for which the external bus must be locked for exclusive use by the microprocessor in processing the sequence of locked transaction requests, the method further comprises the step of locking the external bus for exclusive use by the microprocessor upon issuance of the first split request of the misaligned bus lock transaction request from the sequencer to the control logic.

13. The method of claim 11, wherein for a misaligned bus transaction request comprising a bus unlock transaction request of a sequence of locked transaction requests for which the external bus must be locked for exclusive use by the microprocessor in processing of the sequence of locked transaction requests, the method further comprises the step of unlocking the external bus to enable use of the external bus by other bus agents when the second response information for the second split request received from the external bus agents indicates that no errors have occurred in processing of the second split request.

14. The method of claim 11, wherein the step of identifying bus transaction requests which cross a data bus width boundary of the external bus as misaligned bus transaction requests is performed by one of the bus queue and the sequencer.

15. The method of claim 11, wherein the method further comprises the step of canceling the misaligned bus transaction request upon the occurrence of a hard error.

16. An apparatus for sequencing misaligned bus transaction requests between a microprocessor and external bus agents coupled together via an external bus, each said misaligned bus transaction request forming a memory access crossing a data bus width boundary of the external bus, the apparatus comprising:

a transaction request sequencer for receiving bus transaction requests from the microprocessor and identify-

19

ing bus transaction requests which cross a data bus width boundary of the external bus as misaligned bus transaction requests, the sequencer separating each misaligned bus transaction request into at least first and second split transaction requests, with each split request forming a memory access that does not cross a data bus width boundary of the external bus;

control logic for receiving bus transaction requests from the sequencer and processing bus transaction requests on the external bus; and

wherein in the case of a misaligned bus transaction request, the sequencer issues the first and second split requests in order to the control logic for driving of associated request and address information on the external bus to process the first and second requests, with the second split request being issued from the sequencer to the control logic only after receiving a first global observation for the first split request from external bus agents.

17. The apparatus of claim 16, wherein for a misaligned bus transaction request comprising a bus lock transaction request of a sequence of locked transaction requests for which the external bus must be locked for exclusive use by the microprocessor in processing the sequence of locked transaction requests, the external bus is locked for exclusive use by the microprocessor upon issuance of the first split request of the misaligned bus lock transaction request from the sequencer to the control logic.

18. The apparatus of claim 16, wherein for a misaligned bus transaction request comprising a bus unlock transaction request of a sequence of locked transaction requests for which the external bus must be locked for exclusive use by the microprocessor in processing of the sequence of locked transaction requests, the external bus is unlocked to enable use of the external bus by other bus agents when said second global observation for the second split request from the external bus agents is received by the control logic.

19. In a computer system having a microprocessor, an external bus for communicating with external bus agents

20

comprising at least a memory means; and a bus interface unit having control logic for transferring data between the microprocessor and the external bus agents, an apparatus is provided for sequencing misaligned bus transaction requests between the microprocessor and the external bus agents, each said misaligned bus transaction request forming a memory access crossing a data bus width boundary of the external bus, the apparatus comprising:

a transaction request sequencer for receiving bus transaction requests from the microprocessor and identifying bus transaction requests which cross a data bus width boundary of the external bus as misaligned bus transaction requests, the sequencer separating each misaligned bus transaction request into at least first and second split transaction requests, with each split request forming a memory access that does not cross a data bus width boundary of the external bus;

control logic for processing bus transaction requests issued from the sequencer on the external bus; and

wherein in the case of a misaligned bus transaction request, the sequencer issues the first and second split requests in order to the control logic for driving of associated request and address information on the external bus to process the first and second requests, with the second split request being issued from the sequencer to the control logic only after receiving a first global observation for the first split request from external bus agents.

20. The apparatus of claim 19, wherein the microprocessor comprises an out-of-order microprocessor having an execution unit for executing instructions out-of-order, a dispatch buffer for temporarily storing microinstructions until the execution unit is available and a reorder buffer for buffering instructions and corresponding execution results after instruction execution.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION

PATENT NO. : 5,535,345
DATED : July 9, 1996
INVENTOR(S) : Fisch et al.


It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

In column 4 at line 56 delete "trait" and insert --unit--

In column 15 at line 32 delete "request issuing the second split" and insert --request, issuing the second split--

Signed and Sealed this
Twenty-sixth Day of November 1996

Attest:



BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks



US006175889B1

(12) **United States Patent**
Olorig

(10) **Patent No.:** **US 6,175,889 B1**
(45) **Date of Patent:** **Jan. 16, 2001**

(54) **APPARATUS, METHOD AND SYSTEM FOR A COMPUTER CPU AND MEMORY TO HIGH SPEED PERIPHERAL INTERCONNECT BRIDGE HAVING A PLURALITY OF PHYSICAL BUSES WITH A SINGLE LOGICAL BUS NUMBER**

5,878,237 * 3/1999 Olorig .

* cited by examiner

Primary Examiner—Gopal C. Ray
(74) *Attorney, Agent, or Firm*—Williams, Morgan & Amerson, P.C.

(75) **Inventor:** **Sompong Paul Olorig**, Cypress, TX (US)

(73) **Assignee:** **Compaq Computer Corporation**, Houston, TX (US)

(*) **Notice:** Under 35 U.S.C. 154(b), the term of this patent shall be extended for 0 days.

(21) **Appl. No.:** **09/177,441**

(22) **Filed:** **Oct. 21, 1998**

(51) **Int. Cl.⁷** **G06F 13/40; G06F 13/38**

(52) **U.S. Cl.** **710/129; 710/126; 710/128; 370/402**

(58) **Field of Search** 70/129, 126, 128, 70/101, 107, 113, 62; 370/401, 402; 711/100, 200

(56) **References Cited**

U.S. PATENT DOCUMENTS

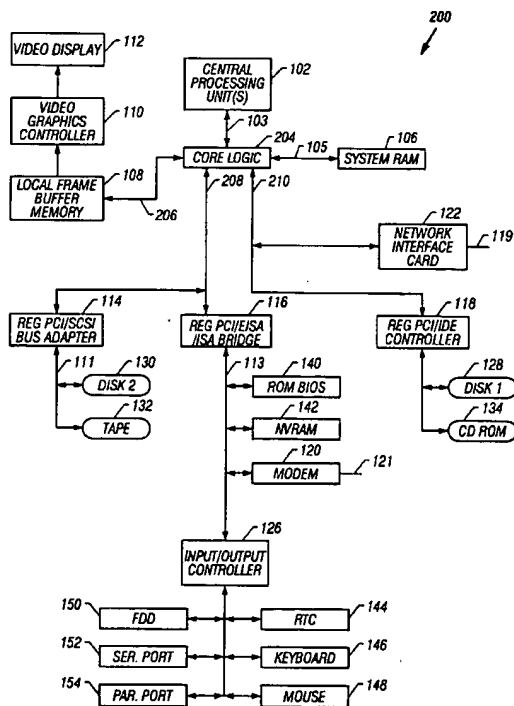
5,734,850 * 3/1998 Kenny et al. .

5,751,975 * 5/1998 Gillespie et al. .

(57) **ABSTRACT**

A core logic chip set in a computer system provides a bridge between processor host and memory buses and a plurality of registered peripheral component interconnect ("PCI-X") buses capable of operating at 66 MHz. Each of the plurality of PCI-X buses have the same logical bus number. The core logic chip set has an arbiter having Request ("REQ") and Grant ("GNT") signal lines for each PCI-X device connected to the plurality of PCI-X physical buses. Each of the plurality of PCI-X buses has its own read and write queues to provide transaction concurrency of PCI-X devices on different ones of the plurality of PCI-X buses when the transaction addresses are not the same or are M byte aligned. Upper and lower memory address range registers store upper and lower memory addresses associated with each PCI-X device. Whenever a transaction occurs, the transaction address is compared with the stored range of memory addresses. If a match between addresses is found then strong ordering is used. If no match is found then weak ordering may be used to improve transaction latency times. PCI-X device to PCI-X device transactions may occur without being starved by CPU host bus to PCI-X bus transactions.

57 Claims, 13 Drawing Sheets



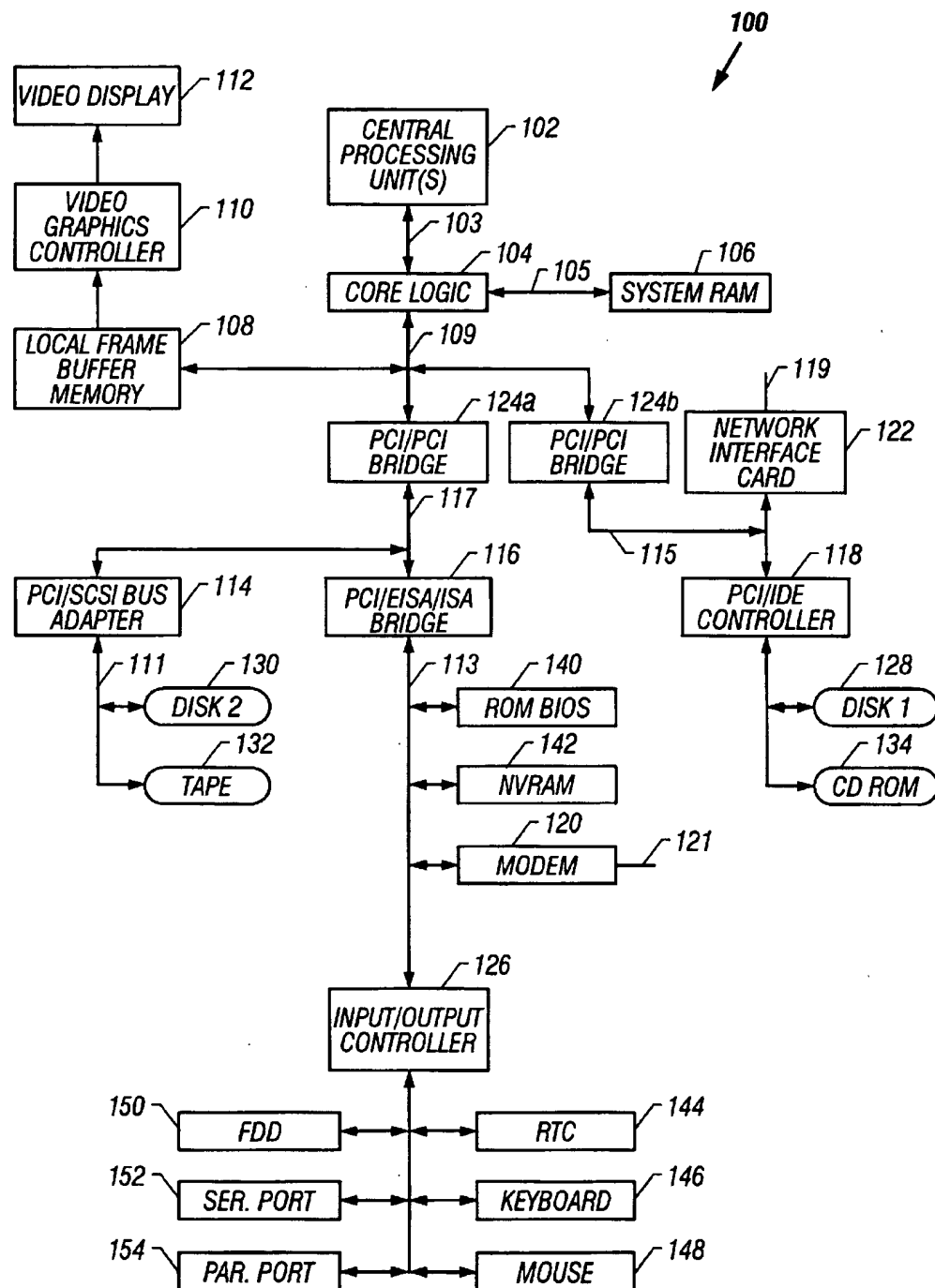


FIG. 1
(Prior Art)

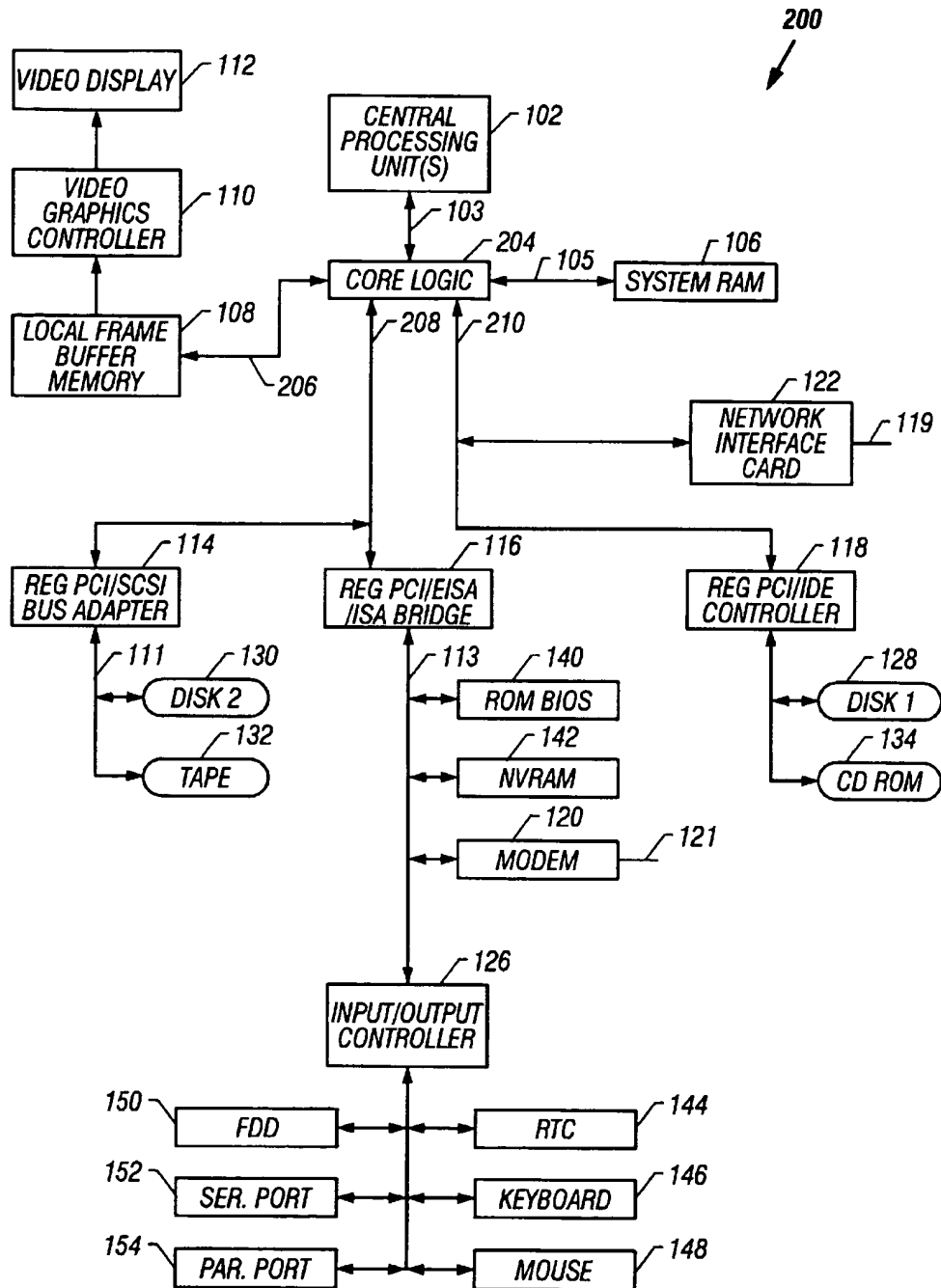


FIG. 2

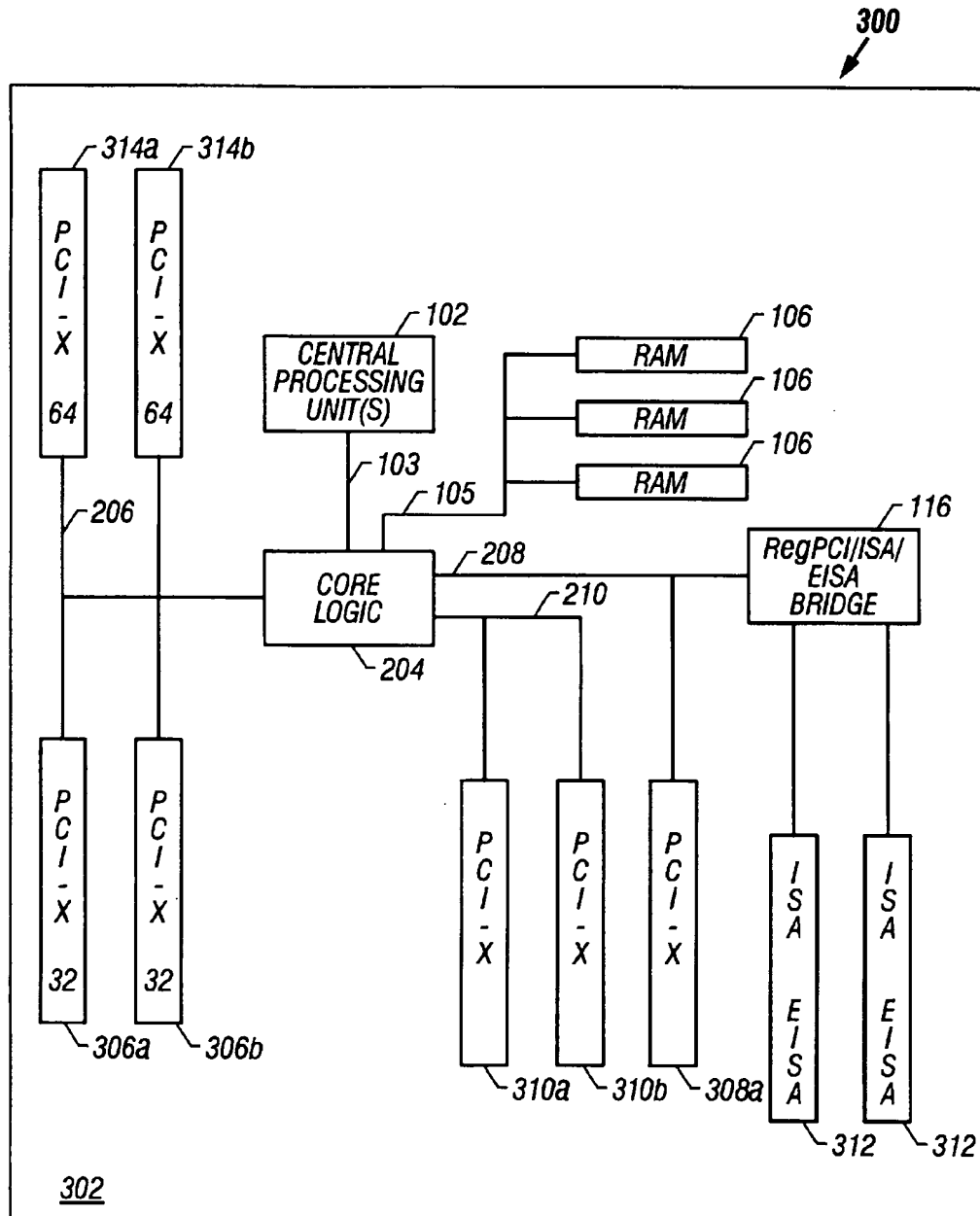


FIG. 3

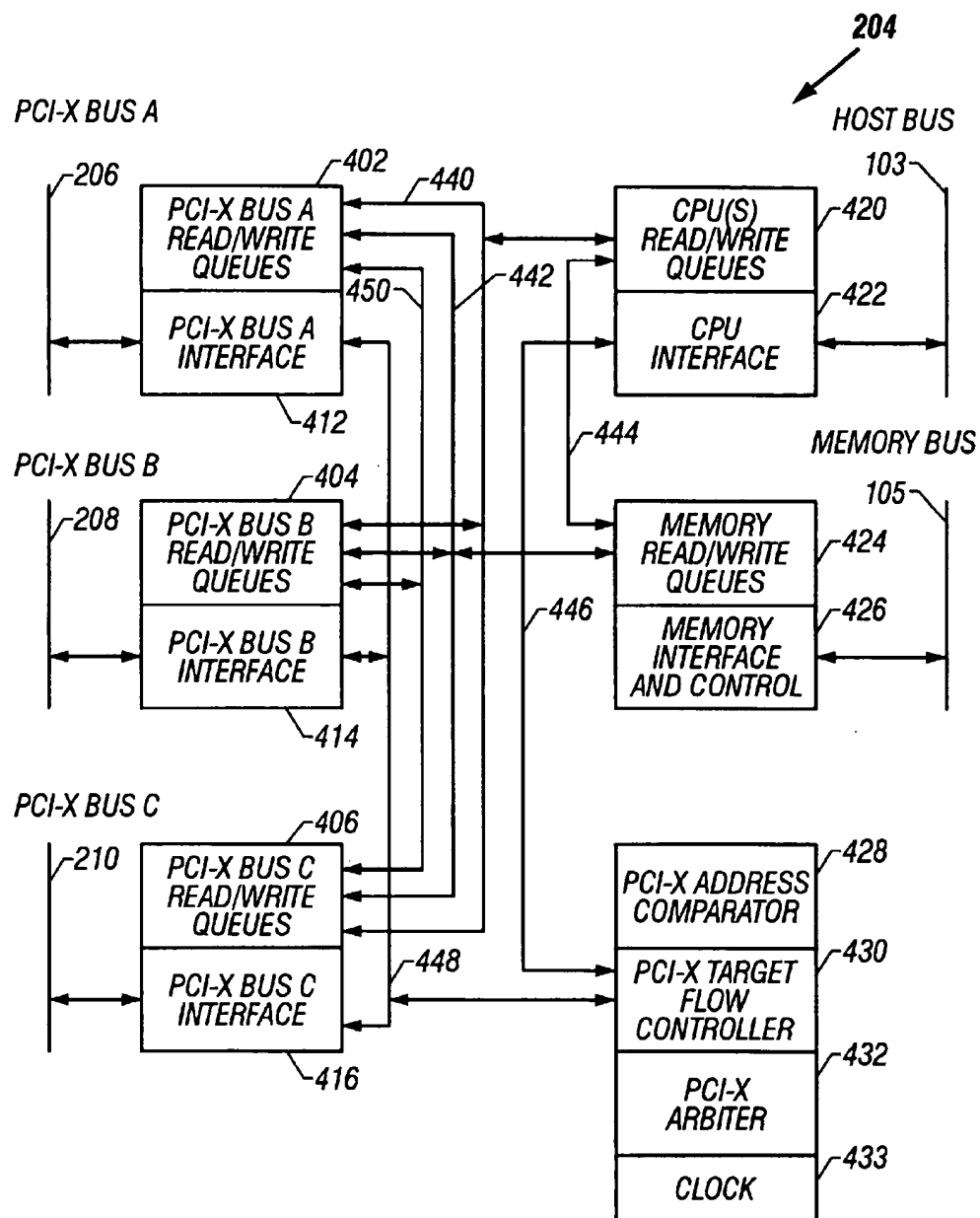
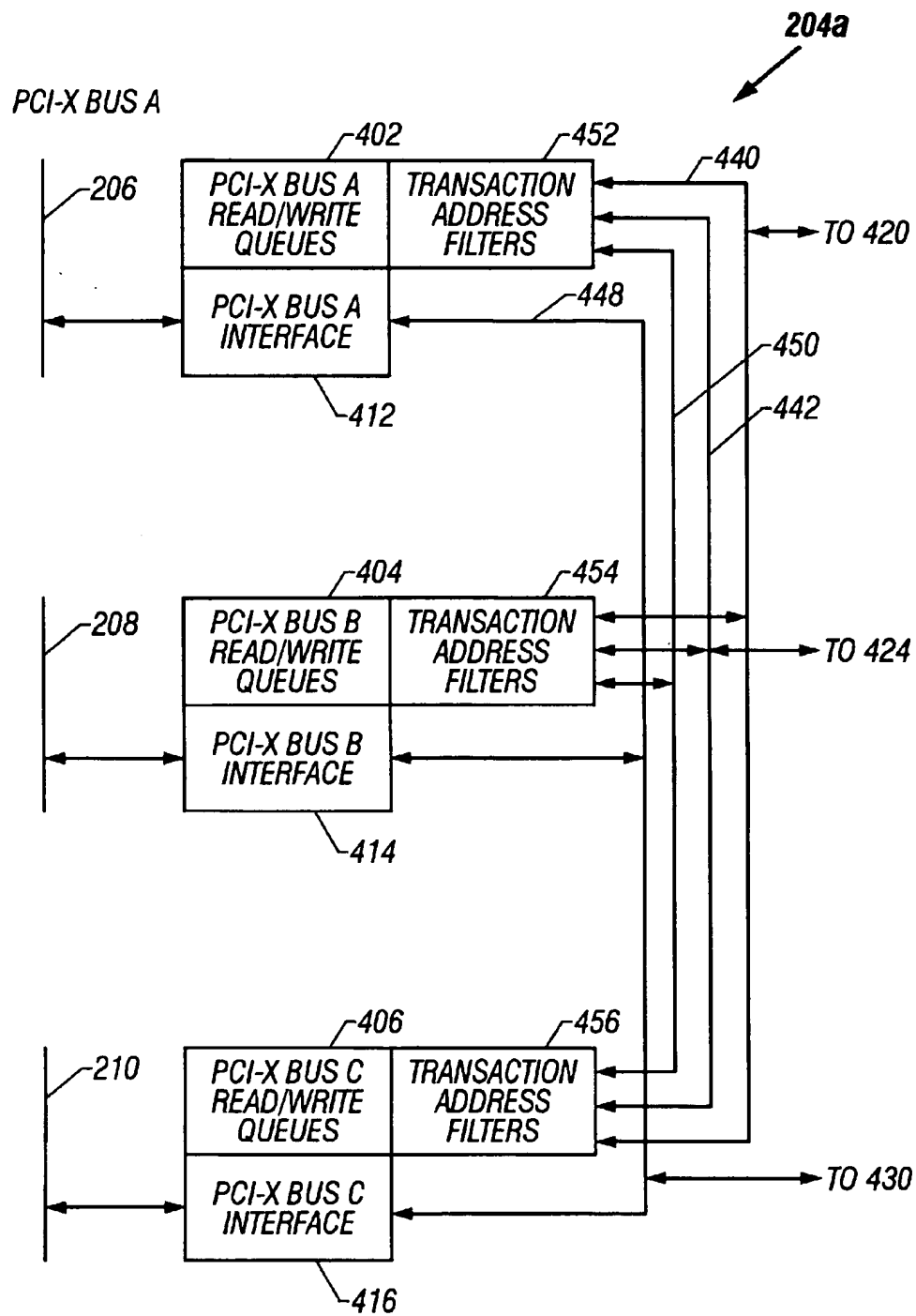
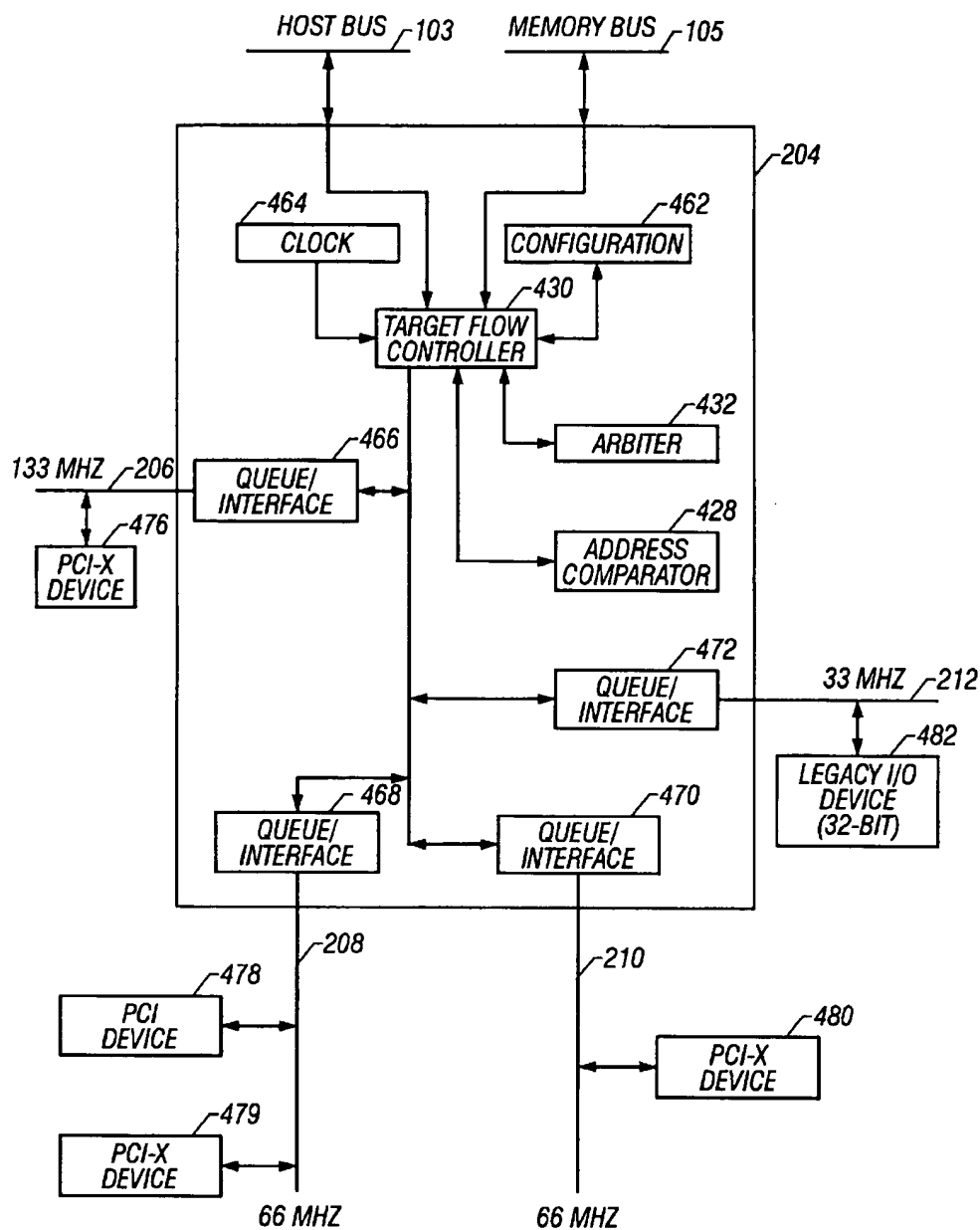


FIG. 4

**FIG. 4A**

**FIG. 4B**

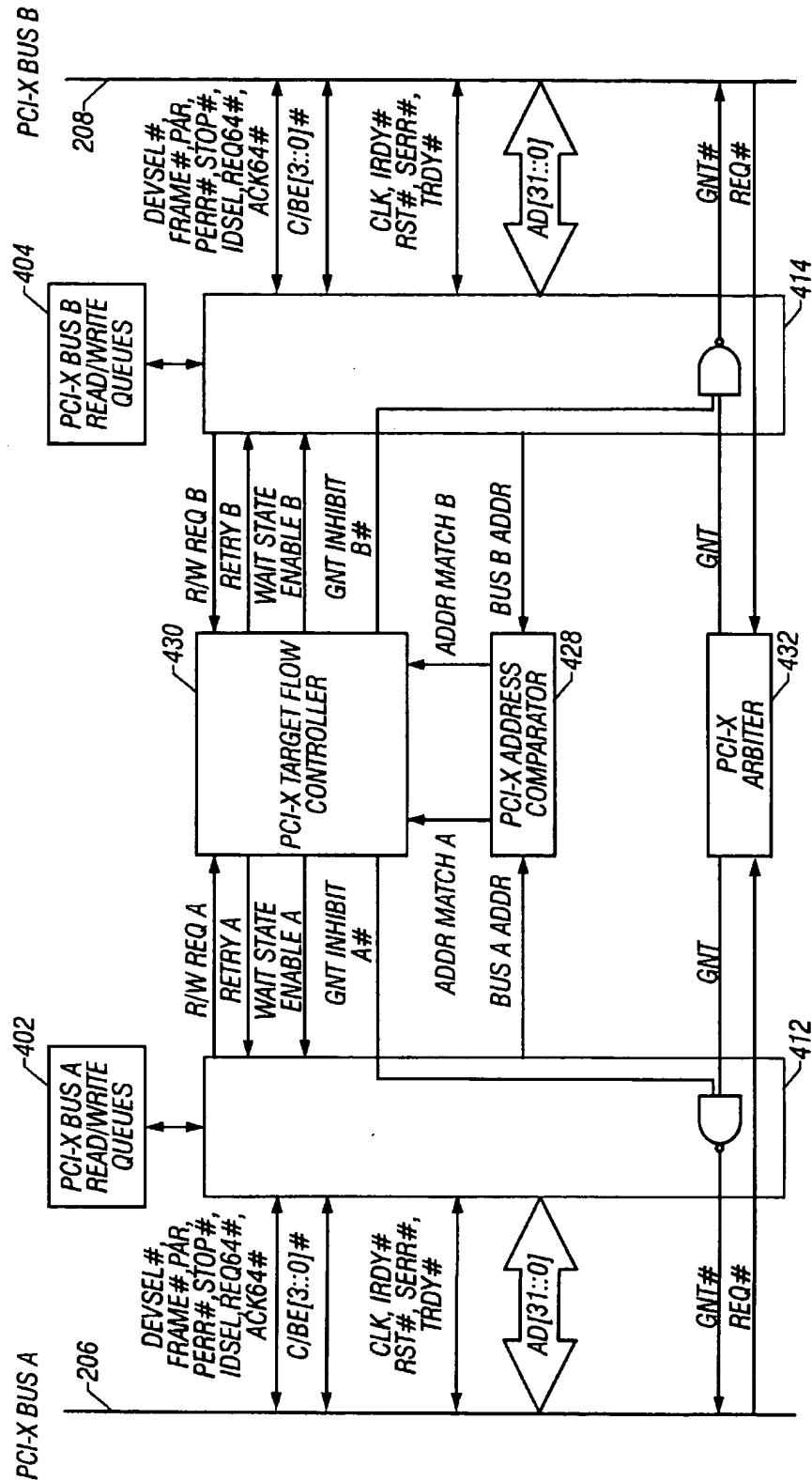


FIG. 5

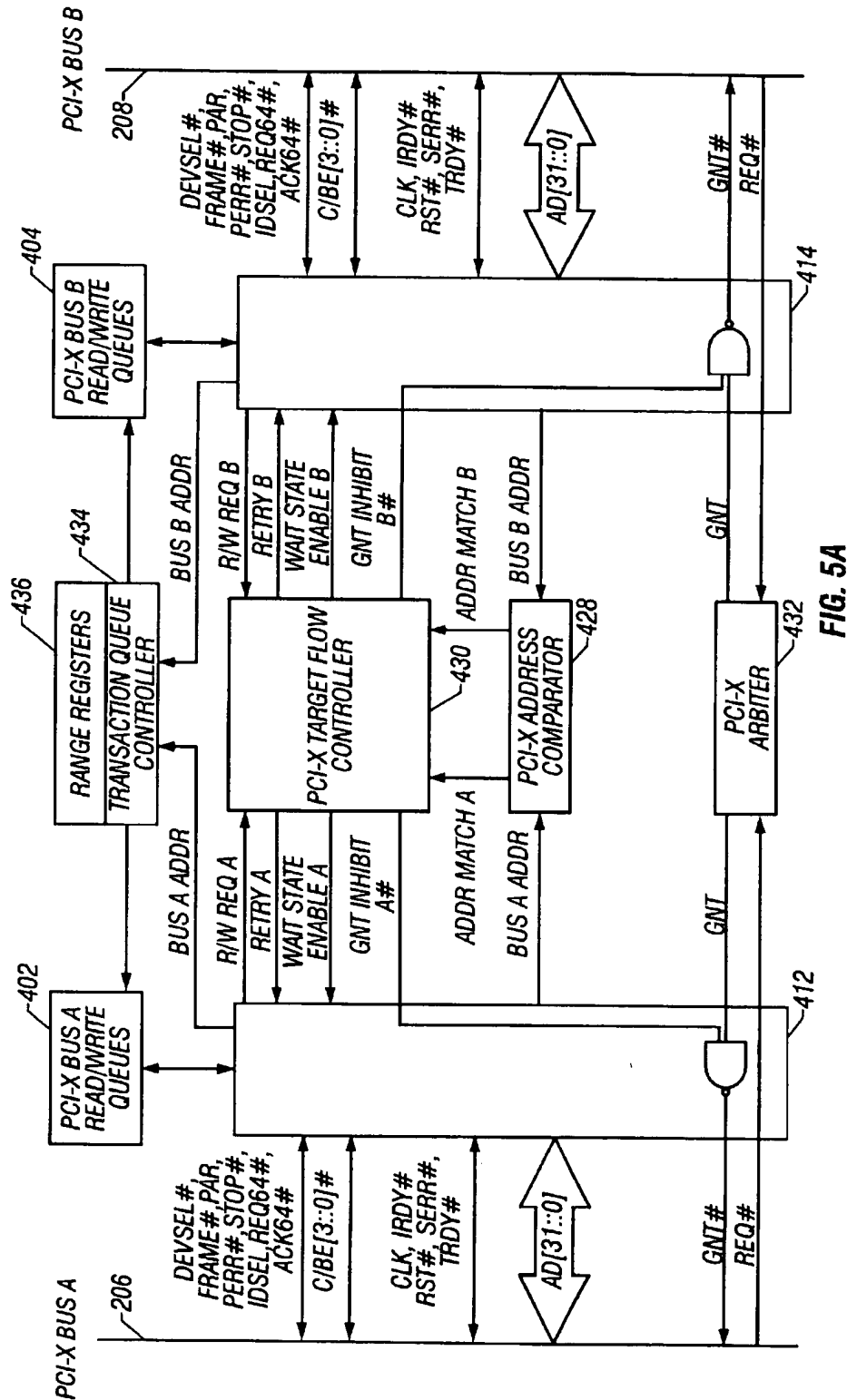
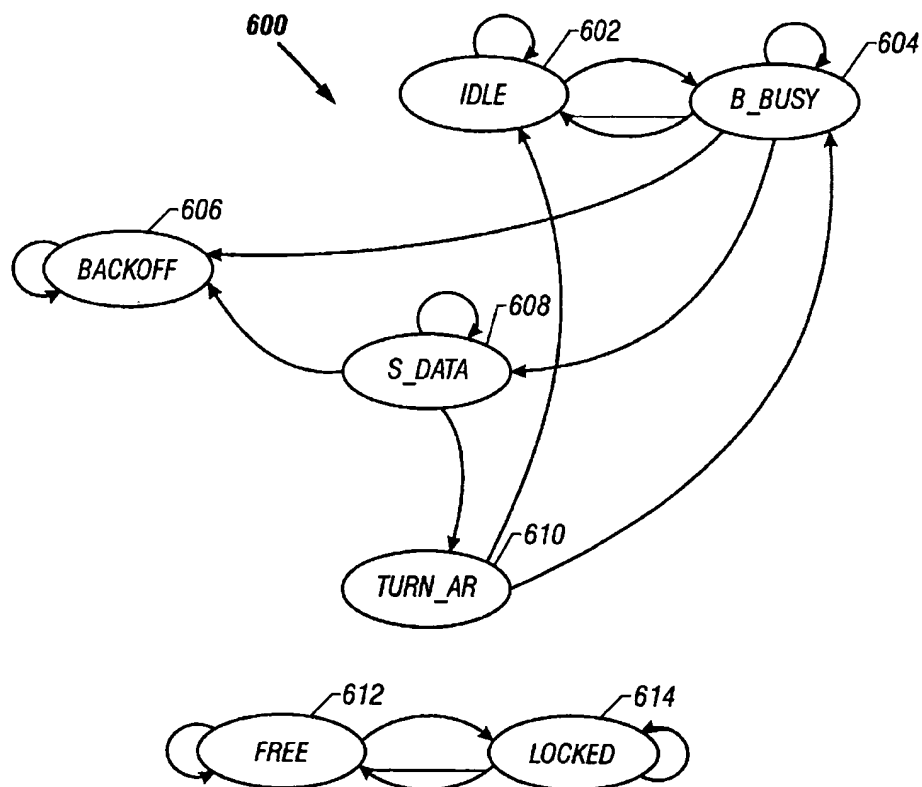
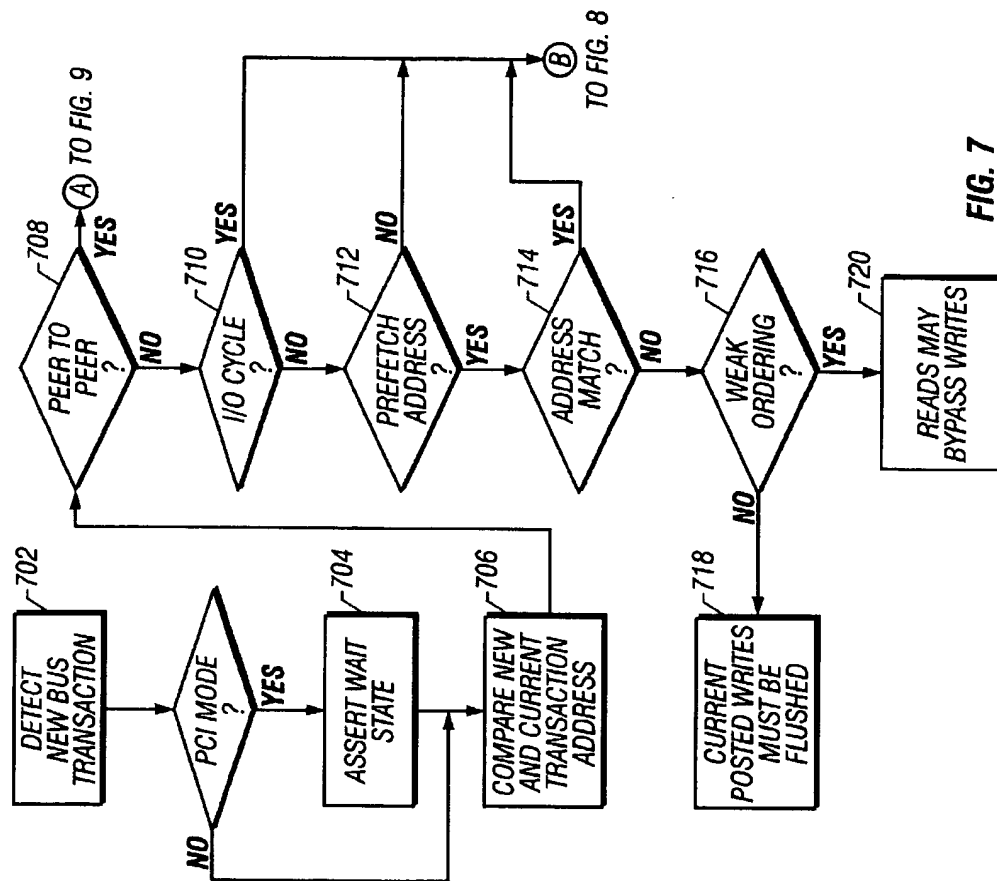


FIG. 5A

436

502	504	506
PCI-X DEVICE A	UPPER ADDR A	LOWER ADDR A
PCI-X DEVICE B	UPPER ADDR B	LOWER ADDR B
PCI-X DEVICE C	UPPER ADDR C	LOWER ADDR C
PCI-X DEVICE D	UPPER ADDR D	LOWER ADDR D
PCI-X DEVICE E	UPPER ADDR E	LOWER ADDR E
•	•	•
•	•	•
•	•	•
PCI-X DEVICE N	UPPER ADDR N	LOWER ADDR N

FIG. 5B**FIG. 6**



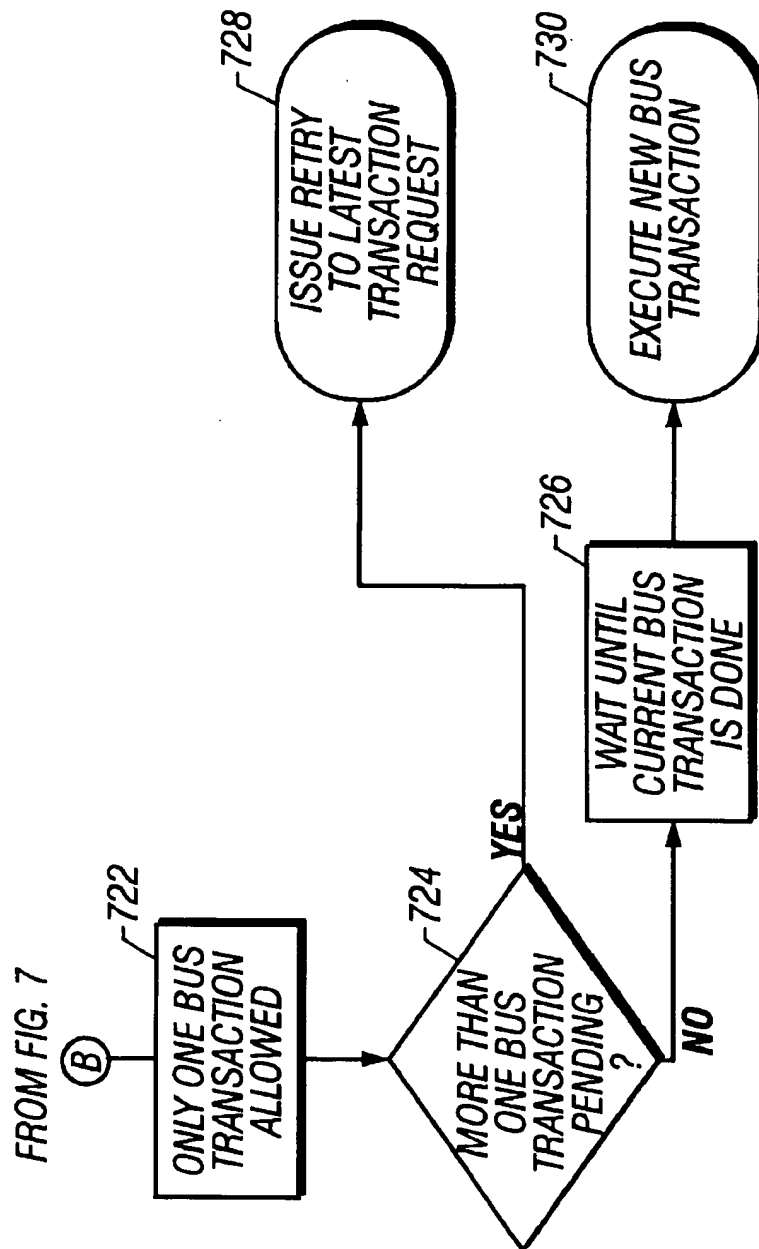


FIG. 8

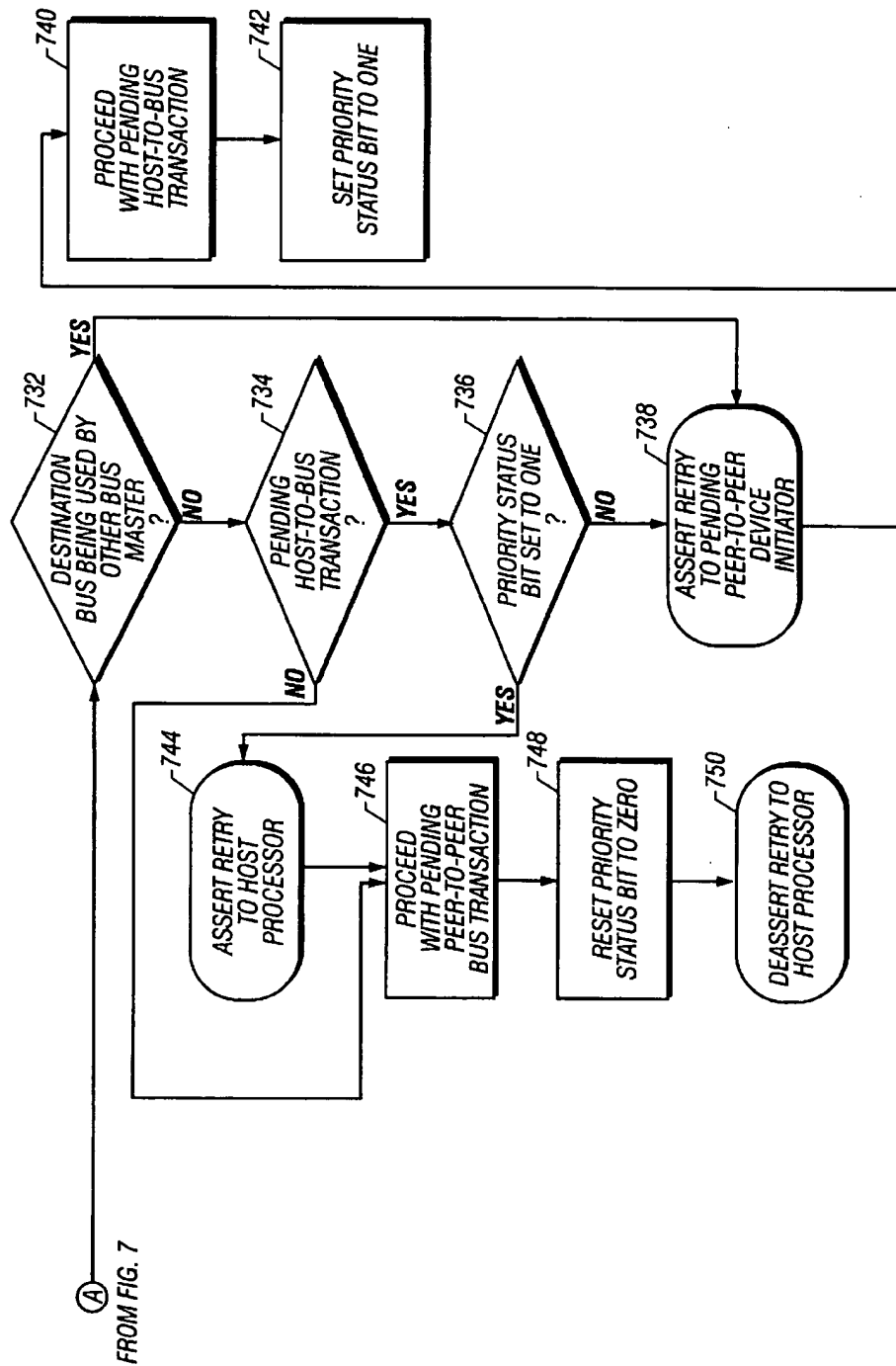
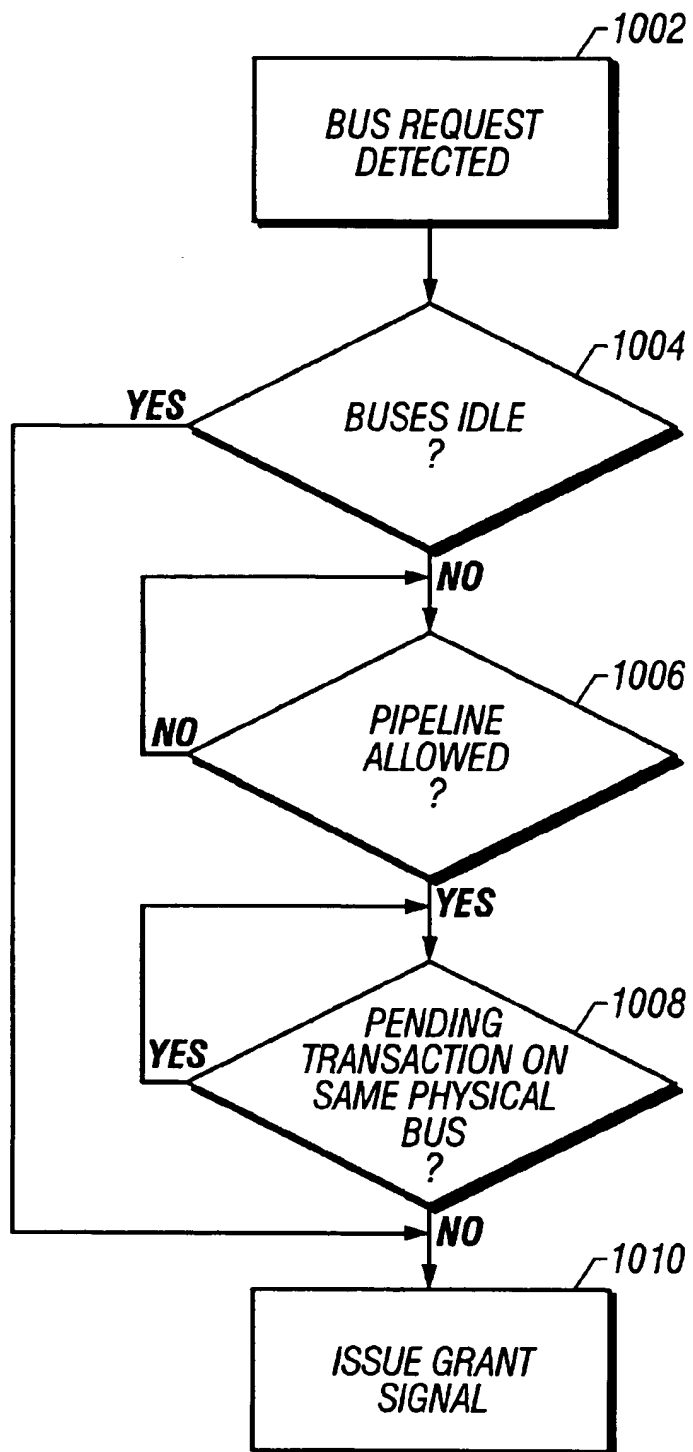


FIG. 9

**FIG. 10**

1

**APPARATUS, METHOD AND SYSTEM FOR A
COMPUTER CPU AND MEMORY TO HIGH
SPEED PERIPHERAL INTERCONNECT
BRIDGE HAVING A PLURALITY OF
PHYSICAL BUSES WITH A SINGLE
LOGICAL BUS NUMBER**

**CROSS REFERENCE TO RELATED PATENT
APPLICATION**

This patent application is related to U.S. patent application Ser. No. 08/853,289; filed May 9, 1997; entitled "Dual Purpose Apparatus, Method And System For Accelerated Graphics Port And Peripheral Component Interconnect" by Ronald T. Horan and Sompong P. Olarig; U.S. patent application Ser. No. 08/893,849, filed Jul. 11, 1997, entitled "Apparatus, Method and System for a Computer CPU and Memory to PCI Bridge Having a Plurality of Physical PCI Buses" by Sompong P. Olarig; U.S. patent application Ser. No. 09/148,042, filed on Sep. 3, 1998, entitled "High Speed Peripheral Interconnect Apparatus, Method And System" by Dwight Riley, Chris Petey, Alan Goodrum, Ryan Callison, Bill Galloway, David Heisey, Thomas Grief, Tim Waldrop and Paul Culley; all of these applications are hereby incorporated by reference for all purposes.

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention relates to computer systems using a bus bridge(s) to interface a central processor(s), random access memory and input-output peripherals together, and more particularly, in utilizing in a computer system a bridge to a plurality of registered peripheral component interconnect (PCI-X) buses wherein the plurality of PCI-X buses have the same logical bus number.

2. Description of the Related Technology

Use of computers, especially personal computers, in business and at home is becoming more and more pervasive because the computer has become an integral tool of most information workers who work in the fields of accounting, law, engineering, insurance, services, sales and the like. Rapid technological improvements in the field of computers have opened up many new applications heretofore unavailable or too expensive for the use of older technology mainframe computers. These personal computers may be used as stand-alone workstations (high-end individual personal computers) or linked together in a network by a "network server" which is also a personal computer that may have a few additional features specific to its purpose in the network. The network server may be used to store massive amounts of data, and may facilitate interaction of the individual workstations connected to the network for electronic mail ("Email"), document databases, video teleconferencing, whiteboarding, integrated enterprise calendar, virtual engineering design and the like. Multiple network servers may also be interconnected by local area networks ("LAN") and wide area networks ("WAN").

A significant part of the ever-increasing popularity of the personal computer, besides its low cost relative to just a few years ago, is its ability to run sophisticated programs and perform many useful and new tasks. Personal computers today may be easily upgraded with new peripheral devices for added flexibility and enhanced performance. A major advance in the performance of personal computers (both workstation and network servers) has been the implementation of sophisticated peripheral devices such as video graphics adapters, local area network interfaces, SCSI bus

2

adapters, full motion video, redundant error checking and correcting disk arrays, and the like. These sophisticated peripheral devices are capable of data transfer rates approaching the native speed of the computer system's microprocessor central processing unit ("CPU"). The peripheral devices' data transfer speeds are achieved by connecting the peripheral devices to the microprocessor(s) and associated system random access memory through high-speed expansion local buses. Most notably, a high-speed expansion local bus standard has emerged that is microprocessor independent and has been embraced by a significant number of peripheral hardware manufacturers and software programmers. This high-speed expansion bus standard is called the "Peripheral Component Interconnect" or "PCI." The complete definition of the PCI local bus may be found in the "PCI Local Bus Specification," revision 2.1; PCI/PCI Bridge Specification, revision 1.0; PCI System Design Guide, revision 1.0; PCI BIOS Specification, revision 2.1, and Engineering Change Notice ("ECN") entitled "Addition of 'New Capabilities' Structure," dated May 20, 1996, the disclosures of which are hereby incorporated by reference for all purposes. These PCI specifications and ECN are available from the PCI Special Interest Group, P.O. Box 14070, Portland, Oreg. 97214.

A computer system has a plurality of information (data and address) buses such as a host bus, a memory bus, at least one high speed expansion local bus such as the PCI bus, and other peripheral buses such as the Small Computer System Interface (SCSI), Extension to Industry Standard Architecture (EISA), and Industry Standard Architecture (ISA). The microprocessor(s) (CPU) of the computer system communicates with main memory and with the peripherals that make up the computer system over these various buses. The microprocessor(s) communicate(s) to the main memory over a host bus to memory bus bridge. The main memory generally communicates over a memory bus through a cache memory bridge to the CPU host bus. The peripherals, depending on their data transfer speed requirements, are connected to the various buses which are connected to the microprocessor host bus through bus bridges that detect required actions, arbitrate, and translate both data and addresses between the various buses.

The choices available for the various computer system bus structures and devices residing on these buses are relatively flexible and may be organized in a number of different ways. One of the more desirable features of present day personal computer systems is their flexibility and ease in implementing custom solutions for users having widely different requirements. Slower peripheral devices may be connected to the ISA or EISA bus(es), other peripheral devices, such as disk and tape drives may be connected to a SCSI bus, and the fastest peripheral devices such as network interface cards (NICs) and video graphics controllers may require connection to the PCI bus. Information transactions on the PCI bus may operate at 33 MHz or 66 MHz clock rates and may be either 32 or 64-bit transactions.

A PCI device may be recognized by its register configuration during system configuration or POST, and the speed of operation of the PCI device may be determined during POST by reading the 66 MHz-CAPABLE bit in the status register, and/or by a hardwired electrical signal "M66EN" as an active "high" input to the 66 MHz PCI device card. If any of the PCI devices on the PCI bus are not 66 MHz capable then the non-66 MHz capable PCI card will deactivate the M66EN signal pin by pulling it to ground reference. If all PCI devices on the PCI bus are 66 MHz capable then M66EN remains active high and each 66 MHz capable PCI card will operate at a 66 MHz bus speed.

The PCI 2.1 Specification supports a high 32-bit bus, referred to as the 64-bit extension to the standard low 32-bit bus. The 64-bit bus provides additional data bandwidth for PCI devices that require it. The high 32-bit extension for 64-bit devices requires an additional 39 signal pins: REQ64#, ACK64#, AD[63:32], C/BE[7:4]#, and PAR64. These signals are defined more fully in the PCI 2.1 Specification incorporated by reference hereinabove. 32-bit PCI devices work unmodified with 64-bit PCI devices. A 64-bit PCI device must default to 32-bit operation unless a 64-bit transaction is negotiated. 64-bit transactions on the PCI bus are dynamically negotiated (once per transaction) between the master and target PCI devices. This is accomplished by the master asserting REQ64# and the target responding to the asserted REQ64# by asserting ACK64#. Once a 64-bit transaction is negotiated, it holds until the end of the transaction. Signals REQ64# and ACK64# are externally pulled up by pull up resistors to ensure proper behavior when mixing 32-bit and 64-bit PCI devices on the PCI bus. A central resource controls the state of REQ64# to inform the 64-bit PCI device that it is connected to a 64-bit bus. If REQ64# is deasserted when RST# is deasserted, the PCI device is not connected to a 64-bit bus. If REQ64# is asserted when RST# is deasserted, the PCI device is connected to a 64-bit bus.

Another advance in the flexibility and ease in the implementation of personal computers is the emerging "plug and play" standard in which each vendor's hardware has unique coding embedded within the peripheral device. Plug and play software in the computer operating system software auto configures the peripheral devices found connected to the various computer buses such as the various PCI buses, EISA and ISA buses. In addition, the plug and play operating system software configures registers within the peripheral devices found in the computer system as to memory space allocation, interrupt priorities and the like.

Plug and play initialization generally is performed with a system configuration program that is run whenever a new device is incorporated into the computer system. Once the configuration program has determined the parameters for each of the devices in the computer system, these parameters may be stored in non-volatile random access memory (NVRAM). An industry standard for storage of both plug and play and non-plug and play device configuration information is the Extended System Configuration Data (ESCD) format. The ESCD format is used to store detailed configuration information in the NVRAM for each device. This ESCD information allows the computer system read only memory (ROM) basic input/output system (BIOS) configuration software to work together with the configuration utilities to provide robust support for all peripheral devices, both plug and play, and non-plug and play.

During the first initialization of a computer, the system configuration utility determines the hardware configuration of the computer system including all peripheral devices connected to the various buses of the computer system. Some user involvement may be required for device interrupt priority and the like. Once the configuration of the computer system is determined, either automatically and/or by user selection of settings, the computer system configuration information is stored in ESCD format in the NVRAM. Thereafter, the system configuration utility need not be run again. This greatly shortens the startup time required for the computer system and does not require the computer system user to have to make any selections for hardware interrupts and the like, as may be required in the system configuration utility.

However, situations often arise which require rerunning the system configuration utility to update the device configuration information stored in the NVRAM when a new device is added to the computer system. One specific situation is when a PCI peripheral device interface card having a PCI—PCI bridge is placed into a PCI connector slot of a first PCI bus of the computer system. The PCI—PCI bridge, which creates a new PCI bus, causes the PCI bus numbers of all subsequent PCI buses to increase by one (PCI—PCI bridge may be a PCI interface card having its own PCI bus for a plurality of PCI devices integrated on the card or for PCI bus connector slots associated with the new PCI bus). This creates a problem since any user configured information such as interrupt request (IRQ) number, etc., stored in the NVRAM specifies the bus and device/function number of the PCI device to which it applies. Originally, this information was determined and stored in the NVRAM by the system configuration utility during the initial setup of the computer system and contains configuration choices made at that time.

During normal startup of the computer system (every time the computer is turned on by the user), a Power On Self Test (POST) routine depends on prior information stored in the NVRAM by the system configuration utility. If the PCI bus numbers of any of the PCI cards change because a new PCI bus was introduced by adding a new PCI—PCI bridge to the computer, the original configuration information stored in the NVRAM will not be correct for those PCI cards now having different bus numbers, even though they remain in the same physical slot numbers. This situation results in the software operating system not being able to configure the PCI cards now having bus numbers different than what was expected from the information stored in the NVRAM. This can be especially bothersome for a PCI device such as a controller which has been configured as a system startup device, but now cannot be used to startup the computer system because its registers have not been initialized during POST to indicate that it is supposed to be the primary controller.

The PCI 2.1 Specification allows two PCI devices on a PCI bus running at 66 MHz. When more than two 66 MHz PCI devices are required in a computer system, a PCI to PCI bus bridge must be added. The PCI to PCI bus bridge is one load, the same as a PCI device card. Thus, adding PCI to PCI bridges is not very efficient when 66 MHz operation of the PCI buses is desired. Each time a PCI to PCI bridge is added to the computer system it creates a new PCI bus having a new PCI bus number. Multiple PCI to PCI bridges running at 66 MHz would typically have to be connected together sequentially, i.e. one downstream from another. Sequentially connecting the PCI to PCI bridges causes increased propagation time and bus to bus handshake and arbitration problems.

PCI devices are connected to the computer system CPU through at least one PCI bus. The at least one PCI bus is in communication with the host bus connected to the CPU through a Host/PCI bus bridge. There exists on the computer system motherboard a set of electrical card edge connector sockets or "slots" adapted to receive one PCI card for each slot. These PCI card slots are numbered as to their physical location on the motherboard and define a unique characteristic for each of the respective PCI card slots and the PCI cards plugged therein. The PCI card slots may be interspersed with other ISA or EISA bus connector slots also located on the computer system motherboard.

The PCI bus closest to the CPU, i.e., the PCI bus just on the other side of the host/PCI bridge is always bus number

zero. Thus, any PCI device card plugged into a PCI slot connected to the number zero PCI bus is defined as being addressable at PCI bus number zero. Each PCI card comprises at least one PCI device that is unique in the computer system. Each PCI device has a plurality of registers containing unique criteria such as Vendor ID, Device ID, Revision ID, Class Code Header Type, etc. Other registers within each PCI device may be read from and written to so as to further coordinate operation of the PCI devices in the computer system. During system configuration, each PCI device is discovered and its personality information such as interrupt request number, bus master priority, latency time and the like are stored in the system non-volatile random access memory (NVRAM) using, for example, the ESCD format.

The number of PCI cards that may be connected to a PCI bus is limited, however, because the PCI bus is configured for high speed data transfers. The PCI specification circumvents this limitation by allowing more than one PCI bus to exist in the computer system. A second PCI bus may be created by connecting another Host-to-PCI bridge to the host bus of the CPU. The second PCI bus connected to the down stream side (PCI bus side) of the second Host-to-PCI bridge is defined as "number one" if there are no other PCI/PCI bridges connected to the PCI bus number zero.

Other PCI buses may be created with the addition of PCI/PCI bridges. For example, a PCI card having a PCI/PCI bridge is plugged into a PCI slot connected to PCI bus number zero on the motherboard of the computer system. In this example, bus number zero is the primary bus because the first host/PCI bridge's PCI bus is always numbered zero. The upstream side of the PCI/PCI bridge is connected to PCI bus number zero and the down stream side of the PCI/PCI bridge now creates another PCI bus which is number one. The prior PCI bus number one on the down stream side of the second Host-to-PCI bus now must change to PCI bus number two. All PCI/PCI bridges connected to or down stream of PCI bus number zero are sequentially numbered. This causes the number of the PCI bus that was created by the second Host-to-PCI bridge to be incremented every time a new PCI bus is created with a PCI/PCI bridge down stream from PCI bus number zero.

When two PCI/PCI bridges are connected to the PCI bus number zero, two PCI buses, numbers one and two, are created. For example, a first PCI card having a PCI/PCI bridge is plugged into motherboard PCI slot number 1, creating PCI bus number one with the PCI/PCI bridge of the first PCI card. A second PCI card having a PCI/PCI bridge is plugged into motherboard PCI slot number 2, creating PCI bus number two with the PCI/PCI bridge of the second PCI card. PCI bus numbers one or two may be connected to PCI devices on the respective first and second PCI cards, or there may be additional PCI card slots on one or both of the first and second PCI cards. When slots are available on a PCI card having a PCI/PCI bridge, additional PCI cards having PCI/PCI bridges may be plugged into the PCI card slots, thus creating more PCI buses. Each PCI/PCI bridge handles information to and from the CPU host bus and a downstream PCI device according to the PCI Specifications referenced above. All embedded PCI devices on the computer system motherboard are assigned a physical slot number of zero (0) and must be differentiated by their respective PCI device and bus numbers.

A computer system may be configured initially with two Host-to-PCI bridges connected to the CPU host bus. This results in the creation of two PCI buses numbered zero and one. These two PCI buses are available for connecting the

PCI devices used in the computer system to the CPU. The system configuration program is run once to establish the personality of each of the PCI devices connected to the two PCI buses, to define interrupt priorities and the like. The configuration information for each of the PCI devices and their associated PCI bus numbers may be stored in the NVRAM using the ESCD format. Thereafter each time the computer system is powered up, the configuration information stored in the NVRAM may be used for initializing and configuring the PCI devices during startup of the operating system and eventually running the application programs.

Initial startup of the computer system is by programs stored in the computer system read only memory (ROM) basic input/output system (BIOS) whose contents may be written into random access memory (RAM) space along with the configuration information stored in the NVRAM so that the computer system may do its startup routines more quickly and then load the operating system software from its hard disk. During the POST routine the computer system depends on the configuration information stored in the NVRAM to access the PCI devices at the PCI bus numbers determined during execution of the original system configuration program.

All of the stored PCI device bus numbers in the NVRAM must match the actual PCI bus numbers for the PCI devices (hard disk SCSI interface, etc.) required during startup of the computer system. If the PCI bus numbers stored in the NVRAM do not match the actual PCI bus numbers, proper computer system operation may be impaired. PCI bus numbers may change if new PCI/PCI bridges are added to the computer system after the configuration program was run to store the system configuration settings in the NVRAM in ESCD format.

Another requirement of the PCI 2.1 Specification is the PCI bridges must follow certain transaction ordering rules to avoid "deadlock" and/or maintain "strong" ordering. To guarantee that the results of one PCI initiator's write transactions are observable by other PCI initiators in the proper order of occurrence, even though the write transactions may be posted in the PCI bridge queues, the following rules must be observed:

- 1) Posted memory writes moving in the same direction through a PCI bridge will complete on the destination bus in the same order they complete on the originating bus;
- 2) Write transactions flowing in one direction through a PCI bridge have no ordering requirements with respect to write transactions flowing in the other direction of the PCI bridge; and
- 3) Posted memory write buffers in both directions must be flushed or drained before starting another read transaction.

Newer types of input-output devices such as "cluster" I/O controllers may not require "strong" ordering but are very sensitive to transaction latency.

Computer system peripheral hardware devices, i.e., hard disks, CD-ROM readers, network interface cards, video graphics controllers, modems and the like, may be supplied by various hardware vendors. These hardware vendors must supply software drivers for their respective peripheral devices used in each computer system even though the peripheral device may plug into a standard PCI bus connector. The number of software drivers required for a peripheral device multiplies for each different computer and operating system. In addition, both the computer vendor, operating system vendor and software driver vendor must test and

certify the many different combinations of peripheral devices and the respective software drivers used with the various computer and operating systems. Whenever a peripheral device or driver is changed or an operating system upgrade is made, retesting and recertification may be necessary.

The demand for peripheral device driver portability between operating systems and host computer systems, combined with increasing requirements for intelligent, distributed input-output ("I/O") processing has led to the development of an "Intelligent Input/Output" ("I₂O") specification. The basic objective of the I₂O specification is to provide an I/O device driver architecture that is independent of both the specific peripheral device being controlled and the host operating system. This is achieved by logically separating the portion of the driver that is responsible for managing the peripheral device from the specific implementation details for the operating system that it serves. By doing so, the part of the driver that manages the peripheral device becomes portable across different computer and operating systems. The I₂O specification also generalizes the nature of communication between the host computer system and peripheral hardware, thus providing processor and bus technology independence. The I₂O specification, entitled "Intelligent I/O (I₂O) Architecture Specification," Draft Revision 1.5, dated March 1997, is available from the I₂O Special Interest Group, 404 Balboa Street, San Francisco, Calif. 94118; the disclosure of this I₂O specification is hereby incorporated by reference.

In the I₂O specification an independent intelligent input-output processor (IOP) is proposed which may be implemented as a PCI device card. The IOP connects to a PCI bus and is capable of performing peer-to-peer PCI transactions with I/O PCI devices residing on the same or other PCI buses. A problem may exist, however, in computer systems having one or more high speed central processing units that perform a plurality of host to PCI transactions. These host to PCI transactions may occur so frequently and quickly that PCI to PCI transactions may be starved due to lack of PCI bus availability.

What is needed is an apparatus, method, and system for a computer that provides a core logic chip set having a bridge for a CPU(s) host bus and random access memory bus to a plurality of PCI buses wherein the plurality of PCI buses have the same logical bus number and are capable of operation at 66 MHz or faster. In addition, a way to determine the strength of write transaction ordering is desired so that maximum advantage may be used to reduce bus transaction latency by taking transactions out of order when these transactions are determined not to require "strong" ordering. Further, a way to prevent PCI-to-PCI transactions from being starved by host-to-PCI transactions is desired.

SUMMARY OF THE INVENTION

The present invention solves the problems inherent in the prior art, at least in part, by providing in a computer system a core logic chip set that is capable of bridging between a CPU(s) host bus, a random access memory bus and a plurality of physical registered PCI buses, wherein the plurality of physical registered PCI buses have the same logical bus number and are capable of operating at 66 MHz or faster. Registered PCI ("PCI-X") buses are comprised of a registered peripheral component interconnect bus, logic circuits therefor, and signal protocols thereof. According to the PCI-X specification, all signals are sampled on the rising edge of the PCI bus clock and only the registered version of

these signals are used inside the PCI-X devices. In the current PCI 2.1 Specification, there are many cases where the state of an input signal setting up to a particular clock edge affects the state of an output signal after that same clock edge. This type of input-output signal behavior is not possible in a registered interface, thus PCI-X introduces the concept of a clock-pair boundary which replaces some single-clock-edges where control signals change. Timing on the PCI-X bus is not as critical as the aforementioned 66 MHz PCI 2.1 Specification, even when the PCI-X bus runs faster than 133 MHz. The PCI-X allows PCI bus operation with more than two PCI device cards.

In the present invention, each of the plurality of physical PCI-X buses may be connected to two PCI-X devices or two PCI-X device card connectors. Thus, each of the plurality of physical PCI-X buses may easily meet the PCI-X Specification for 133 MHz or faster operation. One or more of the physical PCI-X buses also may be connected to more than two PCI-X devices or PCI-X device card connectors and operate at 33 MHz. The plurality of physical PCI-X buses all have the same logical PCI-X bus number of zero since no intermediate PCI-X to PCI-X bridges are needed nor desired in the present invention. Thus, identification of PCI-X devices in the computer system during startup is greatly simplified because there can be no confusion as to which PCI-X bus numbers the PCI-X devices are associated. Configuration information stored in the NVRAM is also simplified since the same PCI-X bus number (zero) is typically associated with each PCI-X device. A PCI-X-to-PCI-X bridge on a multi-function PCI-X device card would create a new PCI-X bus number (only to that card and the multiple PCI-X devices thereon), however, and is contemplated in the present invention. An arbiter(s) in the core logic chip set provides Request ("REQ") and Grant ("GNT") signal lines for each one of the PCI-X devices connected to the plurality of physical PCI-X buses in the computer system. The embodiments of the present invention contemplate a core logic chip set which may be one or more integrated circuit devices such as an Application Specific Integrated Circuit ("ASIC"), Programmable Logic Array ("PLA"), and the like.

The PCI bus is designed to provide connectivity to very high bandwidth devices such as 3-D graphics and gigabit input-output ("I/O") devices. 66 MHz PCI devices are recognized by one static signal which replaces an existing ground pin in the 33 MHz PCI 2.1 Specification, and one bit added to the Configuration Status register as more fully defined in the PCI 2.1 Specification incorporated by reference hereinabove. 66 MHz PCI device bus drivers are basically the same as those used for 33 MHz bus operation but require faster timing parameters and have more critical timing constraints. Thus, the PCI 2.1 Specification recommends only two PCI connector slots for a PCI bus operating at 66 MHz. However, Both 66 MHz and 33 MHz PCI bus operation are contemplated herein for the present invention. However, the PCI-X operation is a compatible superset of the original PCI 2.1 Specification. PCI-X allows for higher clock frequencies such as, for example, 133 MHz in a fully backward-compatible way. PCI-X devices may be designed to meet Registered PCI requirements and still operate as conventional 33 MHz and 66 MHz PCI devices when installed in legacy computer systems. Similarly, if conventional PCI devices are installed in a PCI-X bus, the clock remains at a frequency acceptable to the conventional device, and other devices are restricted to using conventional protocol when communicating with the conventional device. It is expected that this high degree of backward compatibil-

ity will enable the gradual migration of systems and devices to bandwidths in excess of 1 Gbyte/s.

As contemplated herein, the core logic chip set is connected to a plurality of physical PCI-X buses capable of running at 66 MHz or faster. These 66 MHz physical PCI-X buses may also be connected to a combination of PCI (or PCI-X) devices embedded on the computer system motherboard and/or PCI-X device cards plugged into PCI-X card edge connectors also on the computer system motherboard. In the present invention, there is no practical limit to the number of physical PCI-X buses, therefore, as many PCI and PCI-X devices and card slot connectors as needed may be utilized in the computer system. Typically, the number of PCI or PCI-X devices would be limited by the number of Initialization Device Select ("IDSEL") addresses that are available and how the address data lines AD[31:11] are configured on the computer system motherboard to the embedded PCI-X devices and/or the PCI-X card slot connectors. Therefore, the host to PCI-X bus bridge, according to the present invention, will handle transactions with the PCI-X devices as if they were all on the same PCI-X bus.

In the present invention, PCI-X device to PCI-X device transactions are also contemplated herein. When a PCI-X device on a physical PCI-X bus addresses another PCI-X device's memory or I/O addresses on the same physical PCI-X bus or on another physical PCI-X bus, this is referred to hereinafter as "peer-to-peer" PCI bus transactions. It is contemplated in the present invention that peer-to-peer transactions may be enabled or disabled by setting a control register bit in the core logic. The present invention may broadcast the peer-to-peer transaction address onto the physical PCI-X buses so that the intended PCI-X target may respond. Once the target PCI-X device responds, the peer-to-peer transaction is completed. There is no host bus or memory bus activity required for peer-to-peer PCI-X bus transactions. Concurrent transaction activity may occur, however, on other physical PCI-X buses between the memory bus and/or host bus as more fully described hereinafter. This is especially useful when using intelligent, distributed input-output ("I/O") processing as more fully defined in the "Intelligent Input/Output" ("I₂O") specification, entitled "Intelligent I/O (I₂O) Architecture Specification," Draft Revision 1.5, dated March 1997; the disclosure of which is incorporated by reference hereinabove.

In an embodiment of the present invention, the host or memory to PCI-X bus bridge handles requests from PCI-X devices on the different physical PCI-X buses, as mentioned above, just as if they came from the same logical PCI-X bus. This embodiment of the present invention allows only one PCI-X transaction to occur at a time and the arbiter thereof only asserts GNT# to the PCI-X device associated with the current transaction. However, the next PCI-X transaction requested on a different physical PCI-X bus can be pipelined, i.e., the PCI-X device making the next PCI-X bus transaction request may have its GNT# signal issued a few clocks earlier than it could if both of the PCI-X devices were on the same physical PCI-X bus. In this embodiment, the core logic chip set arbiter detects that the current bus master is about to terminate the current transaction or target initiated termination, then and only then will the GNT# signal be issued to the PCI-X device requesting the next transaction. This easily handles PCI-X locked cycles which require the arbiter to wait until the current lock cycle transactions are complete before allowing another PCI-X device transaction to occur.

Another embodiment of the present invention provides in the core logic chip set, separate queues for each of the

plurality of physical PCI-X buses so that PCI-X devices on the different physical PCI-X buses may perform transactions concurrently when the transactions from the different physical PCI-X buses are defined by different memory addresses so long as these memory addresses have been marked as prefetchable. A PCI-X device can mark an address range as prefetchable if there are no side effects on reads, the PCI-X device returns all bytes on reads regardless of the byte enables, and the core logic host bridge can merge processor (s) writes without causing errors, in accordance with the PCI-X Specification.

For example, a PCI-X read transaction is occurring on the physical PCI-X bus A and there is a bus request on the physical PCI-X bus B. The arbiter can assert GNT# to the PCI-X device on the physical PCI-X bus B immediately without waiting for the current transaction to be completed on the physical PCI-X bus A. Once the command and address are valid on the physical PCI-X bus B, the core logic chip set of the present invention inserts at least one wait state to compare the transaction address of physical bus A with the transaction address of physical bus B. If the transaction addresses of physical buses A and B are prefetchable memory addresses, and they are not accessing the same cache-line nor are they M byte aligned, where $M=16 \times 2^n$ and n is 0, 1, 2, 3, 4, etc., the transaction request from the physical PCI-X bus B is allowed to continue until completion. If the transaction addresses are I/O addresses, not prefetchable memory addresses, or physical buses A and B are accessing the same cache-line or are M byte aligned (the transaction addresses from the two physical PCI-X buses overlap the M byte address space), then the transaction request from the physical PCI-X bus B may be delayed by issuing a "retry" to the PCI bus B initiator.

A "retry" is defined in the PCI 2.1 Specification as a termination requested by a target before any data is transferred because the target is busy and temporarily unable to process the transaction. A retry is issued during the first data phase which prevents any data being transferred. Retry is indicated to the initiator by asserting Stop ("STOP#") and deasserting Target Ready ("TRDY#") while keeping Device Select ("DEVSEL#") asserted. This tells the initiator that the target does not intend to transfer the current data item (TRDY# deasserted) and that the initiator must stop the transaction on this data phase (STOP# asserted). The continued assertion of DEVSEL# indicates that the initiator must retry the transaction at a later time (when the commonly addressed transaction on the PCI-X physical bus A has completed). Furthermore, the initiator must use the exact same address, command and byte enables. If it's a write transaction, it must use exactly the same data in the first data phase. The access must be retried until the transaction is completed. The arbitration rules for PCI-X differ from those of standard PCI. Consequently, the bus arbitration rules have been modified for the bridge in those situations where all devices conform to the PCI-X protocol. However, when one or more devices attached to the bridge obeys only the standard PCI protocol, the standard PCI arbitration rules are used.

Another embodiment of the present invention comprises at least two memory address range registers for each PCI-X device in the computer system. At least a top address and a bottom address range register is contemplated for each PCI-X device. The computer system software or application programming interface (API) software for a specific PCI-X device may be used to program the top and bottom address range registers for the specific PCI-X device, i.e., an upper memory address is written into the top address range register

and a lower memory address is written into the bottom address range register. This may be performed during computer system POST or dynamically at any time by different API or applications programs. The range of addresses between the upper and lower memory addresses for each PCI-X device may be used by the present invention in determining whether "strong" or "weak" ordering of PCI-X write transactions are appropriate.

"Strong" ordering requires that the results of one PCI-X initiator's write transactions are observable by other PCI-X initiators in the proper order of occurrence, even though the write transactions may be posted in the PCI-X bridge queues. This is accomplished by the following rules:

- 1) Posted memory writes moving in the same direction through a PCI-X bridge will complete on the destination bus in the same order they complete on the originating bus;
- 2) Write transactions flowing in one direction through a PCI-X bridge have no ordering requirements with respect to write transactions flowing in the other direction of the PCI-X bridge; and
- 3) Posted memory write buffers in both directions must be flushed or drained before starting another read transaction.

These "strong" ordering rules may increase PCI-X bus transaction latency. Newer types of input-output devices such as "cluster" I/O controllers may not require "strong" transaction ordering, but are very sensitive to PCI-X bus transaction latency. According to the present invention, strong ordering for a PCI-X device may be required for a range of memory addresses defined as the upper and lower addresses stored in the respective PCI-X device's top and bottom address range registers. Whenever write transactions are pending that fall within any PCI-X device's current or pending write transactions, then the "strong" ordering rules for bus transactions are appropriate. However, when there are no current or pending write transactions falling within the respective PCI-X device's memory address range requiring strong ordering rules, the present invention may do out-of-order PCI-X transactions, i.e., read transactions may bypass posted write transactions. It is also contemplated herein that additional range registers for each PCI-X device may also define "weak" ordering addresses for the respective PCI-X device. In this way a determination of whether to use "strong" or "weak" transaction ordering rules for current and pending queued transactions may be made.

Another embodiment of the present invention comprises registers which store the I/O and memory address ranges used by the PCI-X devices connected to each physical PCI-X bus. Each PCI-X device is assigned unique memory and/or I/O address ranges by the configuration software. These memory and I/O address ranges are stored in the PCI-X device's configuration registers during initialized at startup (POST). The present invention may also store the memory and I/O address ranges of each PCI-X device connected to a physical PCI-X bus. When a transaction is initiated, the present invention may determine which physical PCI-X bus the target PCI-X device is on by the transaction address. When the transaction address is within an address range associated with a particular physical PCI-X bus, only that PCI-X bus will broadcast the transaction. Thus, only the physical PCI-X bus connected to the intended PCI-X target is activated by the transaction. This feature allows more efficient concurrent transactions within the core logic of the present invention because host-to-PCI-X bus and/or memory-to-PCI-X bus transactions may occur concurrently with the PCI-X-to-PCI-X transactions.

Still another embodiment of the present invention prevents peer-to-peer PCI-X transactions from being starved by repetitive host-to-PCI-X transactions. Host-to-PCI-X transactions may occur rapidly and frequently enough where attempts by one PCI-X device to transact with another PCI-X device is blocked by the higher priority host-to-PCI-X transactions. The present invention solves this problem by allowing at least one PCI-to-PCI-X transaction to occur between host-to-PCI-X transactions. If a PCI-X-to-PCI-X transaction is pending and a host-to-PCI-X transaction is completing or has just completed, the present invention asserts a "retry" signal to the processor host bus if another host-to-PCI-X transaction request is pending. This "retry" signal causes the host processor initiating the host-to-PCI-X transaction request to abort its request and do something else. Later the same host-to-PCI-X transaction request will be initiated again. This allows the processor(s) on the host bus to continue other transactions while the pending PCI-X-to-PCI-X transaction is allowed to proceed. Implementation of this "retry" signal will vary with the type of processor(s) used in the computer system. For example, using the Intel Corp., PENTIUM PRO processor, a "Retry Response" is allowed when DEFER# (with HITM# inactive) is asserted during the SnooP Phase. With this Retry Response, the response agent (the present invention) informs the request agent (host processor) that the transaction must be retried. The "Pentium Pro Family Developer's Manual," ISBN 1-55512-259-0 is available from Intel Corporation, and is incorporated herein by reference. For the Intel Corp., PENTIUM and 80486 processors, a "retry" is when the Backoff input (BOFF#) is asserted to abort all outstanding host bus cycles that have not yet completed. The processor remains in bus hold until BOFF# is deasserted at which time the processor restarts the aborted bus cycle(s) in their entirety. "Pentium and Pentium Pro Processors and Related Products," ISBN 1-5552-251-5 is available from Intel Corporation, and is incorporated herein by reference. Once the current PCI-X-to-PCI-X transaction is underway, i.e., is the last transaction to occur, the "retry" signal is deasserted on the host bus and another host-to-PCI-X transaction request is allowed to occur regardless of whether another PCI-X-to-PCI-X transaction is pending. The present invention thus alternates between host-to-PCI-X transactions and PCI-X-to-PCI-X transactions if both are pending.

An advantage of the present invention is that PCI to PCI bridges are no longer needed to increase PCI-X card slots on the computer system motherboard, thus, multiple delayed transactions and potential deadlock cycles may be avoided. Another advantage is that PCI-X transactions on different physical PCI-X buses may be concurrent if the transaction addresses are different.

Another advantage is that "strong" or "weak" ordering rules may be used for transactions depending on memory address ranges programmed for each PCI-X device.

Another advantage of the present invention is that it allows the computer system to utilize more than two of the higher data throughput (bandwidth) PCI-X devices such as additional video graphics controller cards or high speed NICs by using a plurality of physical PCI-X buses without the problems associated with PCI-to-PCI bridges and the confusion and possible system crashes associated with multiple PCI bus numbers which can easily change when a PCI-to-PCI bridge is added, or PCI device cards are moved from one PCI bus slot to another in the computer.

A feature of the present invention is individual queues for each of the plurality of physical PCI-X buses in the computer system.

13

Another feature is checking the next transaction request address with the current transaction request address by inserting a wait state to the PCI-X device requesting the next transaction so as to compare the current transaction address with the next transaction address to determine if concurrent transactions are appropriate. If the compared addresses do not match nor are M byte aligned, where $M=16 \times 2^n$ and n is 0, 1, 2, 3, 4, etc., concurrent transactions may proceed. If the addresses match or are M byte aligned, then a retry cycle is asserted to the PCI initiator requesting the next transaction.

Still another feature is that out-of-order PCI-X transactions may occur when current or pending transactions are "weak" in relation to one another as determined by "strong" or "weak" ordering address ranges programmed for each PCI-X device.

Yet another feature is to allow a PCI-X-to-PCI-X transaction to occur between host-to-PCI-X transactions.

Other and further objects, features and advantages will be apparent from the following description of presently preferred embodiments of the invention, given for the purpose of disclosure and taken in conjunction with the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a schematic block diagram of a prior art computer system;

FIG. 2 is a schematic block diagram of a computer system according to the present invention;

FIG. 3 is a schematic block diagram in plan view of the present invention according to the computer system of FIG. 2; and

FIG. 4 is a schematic functional block diagram of the present invention according to the computer system of FIGS. 2 and 3;

FIG. 4A is a partial schematic functional block diagram of a further embodiment of the present invention according to FIG. 4;

FIG. 4B is a schematic functional block diagram of the present invention according to FIG. 4;

FIG. 5 is a schematic functional block diagram of a portion of the invention of FIG. 4;

FIG. 5A is a schematic functional block diagram another embodiment of a portion of the invention of FIG. 4;

FIG. 5B is a schematic representation of range registers in a portion of an embodiment of the invention according to FIG. 5A;

FIG. 6 is a schematic state diagram of the present invention; and

FIGS. 7, 8, 9 and 10 are process flow diagrams of the present invention.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention is an apparatus, method and system for providing a core logic chip set in a computer system capable of implementing a bridge between host processor and memory buses and a plurality of registered peripheral component interconnect (PCI-X) buses wherein the plurality of PCI-X buses all have the same logical bus number. The present invention is capable of supporting all features of the PCI-X protocol. In addition, the present invention is capable of mixed mode operation, wherein a mix of PCI-only and PCI-X compatible devices operate concurrently. Computer systems utilizing the present invention experience less

14

latency with multiple PCI-X compatible devices than prior art bridges and configurations.

The PCI bus was developed to have sufficient data bandwidth for high performance peripherals such as a video controller, a high speed network interface card(s), a hard disk controller(s), a SCSI adapter, a wide area network digital router, and the like. A PCI bus running at 33 MHz may have a plurality of card connectors attached thereto, however, when the PCI bus runs at 66 MHz the number of card connectors is limited to two because of timing constraints of the digital control signals. Sophisticated graphics and increased network data transfer requirements have put upward pressure on the PCI buses for faster data transfers between the computer system main memory, host processor(s), peripherals and data from other computers on the network. Thus, 66 MHz operation is preferred, and in some cases mandatory, however, a plurality of PCI-to-PCI bus bridges are required to provide enough PCI device card slots for a typical computer system such as a network server or graphics workstation. PCI-to-PCI bus bridges create new PCI bus numbers and introduce increasingly complex data protocol and handshake requirements, multiple delayed transactions, additional bus latency, and potential deadlock cycles.

Registered PCI ("PCI-X") buses are comprised of a registered peripheral component interconnect bus, logic circuits therefor, and signal protocols thereof. According to the PCI-X Specification, all signals are sampled on the rising edge of the PCI bus clock and only the registered version of these signals are used inside the PCI-X devices. In the current PCI 2.1 Specification, there are many cases where the state of an input signal setting up to a particular clock edge affects the state of an output signal after that same clock edge. This type of input-output signal behavior is not possible in a registered interface, thus PCI-X introduces the concept of a clock-pair boundary which replaces some single-clock-edges where control signals change. Timing on the PCI-X bus is not as critical as the aforementioned 66 MHz PCI 2.1 Specification, even when the PCI-X bus runs faster than 133 MHz. The PCI-X allows PCI bus operation with more than two PCI device cards.

The arbitration rules for PCI-X differ from those of standard PCI. Consequently, the bus arbitration rules have been modified for the bridge in those situations where all devices conform to the PCI-X protocol. However, when one or more devices attached to the bridge obey only the standard PCI protocol, all transactions are conducted according to the standard PCI protocol, even if the devices involved are PCI-X capable. The standard PCI bus protocol rules can be found in the PCI Specification cited earlier. The PCI-X bus arbitration rules are as follows:

1. All REQ# and GNT# signals are registered by the arbiter as well as by all initiators.
2. An initiator is permitted to start a new transaction (drive the AD bus, etc.) on any clock N in which the initiator's GNT# was asserted on clock N-2, and when any of the following three conditions are satisfied:
 - a) The bus was idle (FRAME# and IRDY# were both deasserted) on clock N-2.
 - b) The previous transaction was a byte-count transaction, the command was not a reserved command, the extended command was not a reserved validated extended command, and FRAME# was deasserted on clock N-3.
 - c) The previous transaction was a byte-enable transaction, the command was not a reserved

15

command, the extended command was not a reserved validated extended command, and TRDY# or STOP# was asserted on clock N-3.

3. An initiator is permitted to start a new transaction on clock N even if GNT# is deasserted on clock N-1 (assuming the requirements of item 2 above are met).
4. An initiator is permitted to assert and deassert REQ# on any clock. There is no requirement to deassert REQ# after a target termination (STOP# asserted). The arbiter is assumed to monitor bus transactions to determine when a transaction has been target terminated, if the arbiter uses this information to advance to the next bus owner.
5. If all the GNT# signals are deasserted, the arbiter is permitted to assert any GNT# on any clock. After the arbiter asserts GNT# the arbiter can deassert it on any clock. However, the arbiter must fairly provide opportunities for all devices to execute Configuration transactions, which require GNT# to remain asserted for a minimum of five clocks while the bus is idle.
6. If the arbiter deasserts GNT# to one device, it cannot assert GNT# to another device until the next clock.
7. In PCI hot-plug systems, the arbiter must coordinate with the Hot-Plug Controller to prevent hot-plug operations from interfering with other bus transactions.
8. The default Latency Timer value for initiators in PCI-X mode is 31. Configuration software is discouraged from changing the default value.

For illustrative purposes, prior art references and preferred embodiments of the present invention are described hereinafter for computer systems utilizing the Intel x86 microprocessor architecture and certain terms and references will be specific to that processor platform. PCI-X is an interface standard, however, that is hardware independent and may be utilized with any host computer designed for this interface standard. It will be appreciated by those skilled in the art of computer systems that the present invention may be adapted and applied to any computer platform utilizing the PCI-X interface standard including those systems utilizing the Windows, UNIX, OS/2 and Linux operating systems. The PCI-X specification is incorporated U.S. patent application Ser. No. 09/148,042, filed on Sep. 3, 1998 incorporated by reference herein.

Referring now to the drawings, the details of preferred embodiments of the present invention are schematically illustrated. Like elements in the drawings will be represented by like numbers, and similar elements will be represented by like numbers with a different lower case letter suffix.

Referring to FIG. 1, a schematic block diagram of a prior art computer system utilizing multiple PCI buses is illustrated. The prior art computer system is generally indicated by the numeral 100 and comprises a central processing unit(s) ("CPU") 102, core logic 104, system random access memory ("RAM") 106, a video graphics controller 110, a local frame buffer 108, a video display 112, a PCI/SCSI bus adapter 114, a PCI/EISA/ISA bridge 116, a PCI/IDE controller 118, and PCI/PCI bus bridges 124a, 124b. Single or multilevel cache memory (not illustrated) may also be included in the computer system 100 according to the current art of microprocessor computer systems. The CPU 102 may be a plurality of CPUs 102 in a symmetric or asymmetric multi-processor configuration.

The CPU(s) 102 is connected to the core logic 104 through a CPU host bus 103. The system RAM 106 is connected to the core logic 104 through a memory bus 105.

16

The core logic 104 includes a host-to-PCI bridge between the host bus 103, the memory bus 105 and a first PCI bus 109. The local frame buffer 108 is connected between the video graphics controller 110 and the first PCI bus 109. PCI/PCI bridges 124a, 124b are connected to the first PCI bus 109. The PCI/SCSI bus adapter 114 and PCI/EISA/ISA bridge 116 are connected to the PCI-X/PCI-X bridge 124a through a second PCI bus 117. The PCI/IDE controller 118 and a network interface card ("NIC") 122 are connected to the PCI/PCI bridge 124b through a third PCI bus 115. Some of the PCI devices such as the Video controller 110 and NIC 122 may plug into PCI connectors on the computer system 100 motherboard (not illustrated). Three PCI buses 109, 117 and 115 are illustrated in FIG. 1, and may have logical PCI bus numbers of zero, one and two, respectively.

Hard disk 130 and tape drive 132 are connected to the PCI-X/SCSI bus adapter 114 through a SCSI bus 111. The NIC 122 is connected to a local area network 119. The PCI-X/EISA/ISA bridge 116 connects over an EISA/ISA bus 113 to a ROM BIOS 140, non-volatile random access memory (NVRAM) 142, modem 120, and input-output controller 126. The modem 120 connects to a telephone line 121. The input-output controller 126 interfaces with a keyboard 146, real time clock (RTC) 144, mouse 148, floppy disk drive ("FDD") 150, and serial/parallel ports 152, 154. The EISA/ISA bus 113 is a slower information bus than the PCI bus 109, but it costs less to interface with the EISA/ISA bus 113.

When the computer system 100 is first turned on, start-up information stored in the ROM BIOS 140 is used to begin operation thereof. Basic setup instructions are stored in the ROM BIOS 140 so that the computer system 100 can load more complex operating system software from a memory storage device such as the disk 130. Before the operating system software can be loaded, however, certain hardware in the computer system 100 must be configured to properly transfer information from the disk 130 to the CPU 102. In the computer system 100 illustrated in FIG. 1, the PCI-X/SCSI bus adapter 114 must be configured to respond to commands from the CPU 102 over the PCI buses 109 and 117, and transfer information from the disk 130 to the CPU 102 via buses 117, 109 and 103. The PCI/SCSI bus adapter 114 is a PCI device and remains platform independent. Therefore, separate hardware independent commands are used to setup and control any PCI device in the computer system 100. These hardware independent commands, however, are located in a PCI BIOS contained in the computer system ROM BIOS 140. The PCI BIOS is firmware that is hardware specific but meets the general PCI specification. Plug and play, and PCI devices in the computer system are detected and configured when a system configuration program is executed. The results of the plug and play, and PCI device configurations are stored in the NVRAM 142 for later use by the startup programs in the ROM BIOS 140 (PCI BIOS) which configure the necessary computer system 100 devices during startup.

Referring now to FIG. 2, a schematic block diagram of a computer system utilizing the present invention is illustrated. The computer system, according to the present invention, is generally indicated by the numeral 200. Some of the general functions, components and signal paths not dealing with the present invention are the same as in the computer system 100 (FIG. 1), describe above. Noticeably absent from the computer system 200, however, are the PCI-X/PCI-X bridges 124a, 124b. Instead of requiring multiple PCI-X/PCI-X bridges for a plurality of 66 MHz PCI-X devices, the present invention utilizes a plurality of PCI-X

17

bus interfaces in the core logic 204 which are connected to physical PCI-X buses 206, 208, 210. The PCI-X buses 206, 208, 210 are capable of operation at 66 MHz using both 32-bit and 64-bit transactions, as more fully defined in the PCI-X Specification incorporated by referenced above.

The video graphics controller 110 is connected to the local frame buffer memory 108 which is connected to the core logic 204 through the PCI-X bus 206. The PCI-X/SCSI bus adapter 114 and PCI-X/EISA/ISA bridge 116 are connected to the core logic 204 through the PCI-X bus 208. The PCI-X/IDE controller 118 and a network interface card ("NIC") 122 are connected to the core logic 204 through the PCI-X bus 210. All of the remaining components of the computer system 200 are connected and operate the same as the components of the above mentioned computer system 100 (FIG. 1). The PCI-X buses 206, 208, 210 are physically separate PCI-X buses capable of independent concurrent transactions but appear to the computer S startup and operating system software as one logic PCI-X bus. This greatly simplifies keeping track of PCI-X devices connected in the computer system since all PCI-X devices are on only one logical PCI-X bus, and each PCI-X device has its own unique device number (e.g., 1-16). No longer does the computer system software need to remember which PCI-X device is on which PCI-X bus number, nor is there ever a possibility that a PCI-X device bus number will change, causing a system startup problem. Furthermore, no PCI-X-PCI-X bridge type one configuration transactions take place. The present invention greatly simplifies and speeds up recognition of the PCI-X devices in the computer system and improves transaction times thereof for the computer system 200.

Referring now to FIG. 3, a schematic diagram of a computer system motherboard according to FIG. 2 is illustrated in plan view. The computer system motherboard 300 comprises printed circuit board 302 on which components and connectors are mounted thereto. The printed circuit board 302 comprises conductive printed wiring which is used to interconnect the components and connectors thereon. The conductive printed wiring (illustrated as buses 103, 105, 206, 208 and 210) may be arranged into signal buses having controlled impedance characteristics. On the printed circuit board is the core logic 204, CPU(s) 102, RAM 106, PCI-X/ISA/EISA bridge 116, ISA/EISA connectors 312, 66 MHz, 32-bit PCI-X connector 308a (PCI-X physical bus 208), PCI-X connectors 310a, 310b (PCI-X physical bus 210), PCI-X connectors 306a, 306b and 64-bit PCI-X connectors 314a, 314b (PCI-X physical bus 206). The connectors 314a, 314b carry the additional signals required for 64-bit data width PCI-X operation. Either or both 32-bit and 64-bit data width, 66 MHz PCI-X buses are contemplated herein.

Referring now to FIG. 4, a schematic functional block diagram of the present invention according to the computer system of FIGS. 2 and 3 is illustrated. The core logic 204 comprises PCI-X read/write queues 402, 404, 406; CPU read/write queues 420, memory read/write queues 424, PCI-X bus interfaces 412, 414, 416; CPU interface 422, memory interface and control 426, PCI-X address comparator 428, PCI-X target flow controller 430, and PCI-X arbiter 432. In the preferred embodiment of the present invention, a clock 433 is included to synchronize data transmissions among the various devices. Such a clock could be, for example, a phase locked loop (PLL) clock.

Address, data and control information is transferred between the PCI-X read/write queues 402, 404, 406 and the CPU read/write queues 420 over internal bus 440, between

18

the memory read/write queues 424 over internal bus 442, between the PCI-X read/write queues 402, 404, 406 over internal bus 450, and between the memory read/write queues 424 and CPU read/write queues 420 over internal bus 444.

The PCI-X bus interfaces 412, 414, 416 are connected to the PCI-X buses 206, 208, 210, respectively, and transfer information to and from the PCI-X read/write queues 402, 404, 406. The CPU interface 422 is connected to the CPU host bus 103 and transfers information to and from the CPU read/write queues 420. The memory interface and control 426 is connected to the memory bus 105 and transfers information to and from the memory read/write queues 424.

The PCI-X read/write queues 402, 404, 406 in conjunction with the physically separate PCI-X buses 206, 208, 210 allow independent concurrent transactions for PCI-X devices on these buses. The PCI-X address comparator 428 monitors transaction addresses on each of the PCI-X buses 206, 208, 210 and compares the transaction addresses for each of these PCI-X buses to determine if the PCI-X devices (each on a separate PCI-X bus) are trying to access the same cache-line or M byte aligned, where $M=16 \times 2^n$ and n is 0, 1, 2, 3, 4, etc. Unlike the standard PCI protocol, no wait states are required in a pure PCI-X bridge configuration (i.e., where all devices connected to the bridge are PCI-X compatible). However, where one or more devices are not PCI-X compatible, at least one wait state may be inserted in the PCI bus transaction timing cycles by the target flow controller 430 so that sufficient time is available to compare the different physical bus transaction addresses with one another. If the transaction address comparison indicates no M byte aligned address commonality, then concurrent transactions from the different physical PCI-X buses are allowed to proceed over the internal buses 440, 442, 450. However, if there is any M byte aligned address commonality between the transactions then a "retry" is issued to the second PCI-X device having the M byte aligned address request. This is necessary if either or both of the PCI-X devices on the separate physical PCI-X buses 206, 208, 210 is performing or wants to perform a write transaction. When only read transactions are being performed or requested to be performed by the PCI-X devices, then byte aligned addresses are irrelevant to concurrent transactions among the separate physical PCI-X buses if the addresses are prefetchable.

PCI-X device to PCI-X device transactions may occur directly between the PCI-X read/write queues 402, 404, 406 over internal bus 450. When a PCI-X device on a physical PCI-X bus addresses another PCI-X device's memory or I/O addresses on the same physical PCI-X bus or on another physical PCI bus, this is referred to hereinafter as "peer-to-peer" PCI-X bus transactions. Peer-to-peer transactions may be enabled or disabled by setting a control register bit (not illustrated) in the core logic 204. The present invention may broadcast the peer-to-peer transaction address to all of the physical PCI-X buses 206, 208, 210 so that the intended PCI-X target may respond. Once the target PCI-X device responds, the peer-to-peer transaction has been negotiated and will complete according to the PCI-X Specification. The CPU read/write queues 420, or memory read/write queues 424 are not required for peer-to-peer PCI-X bus transactions. Concurrent transaction activity may occur, however, on other physical PCI-X buses between the memory bus 105 and/or host bus 103 as described herein. This is especially useful when using intelligent, distributed input-output ("I/O") processing operating system software as more fully defined in the "Intelligent Input/Output" ("I₂O") specification, entitled "Intelligent I/O (I₂O) Architecture Specification," Draft Revision 1.5, dated March 1997, incorporated by reference hereinabove.

Referring now to FIG. 4A, a partial schematic functional block diagram of a further embodiment of the invention of FIG. 4 is illustrated. The core logic 204a is similar to the core logic 204 illustrated in FIG. 4, but with the addition of transaction address filters 452, 454, 456 connected between the PCI-X read/write queues 402, 404, 406, respectively, and the internal buses 440, 442, 450. The purpose of the transaction address filters 452, 454, 456 is to allow only those PCI-X transactions intended for the specific PCI-X devices connected to the PCI-X buses 206, 208, 210, respectively. The transaction address filters 452, 454, 456 may comprise registers or other means for storage (not illustrated) which store the memory and I/O address ranges of each PCI-X device connected to the respective PCI-X bus (206, 208, 210), and logic which only allows transactions having addresses which fall within these address ranges to pass to the respective PCI-X read/write queues 402, 404, 406. The transaction address filters 452, 454, 456, thus prevent unnecessary PCI-X bus traffic on the physical buses not connected to the intended target PCI-X devices. This may reduce transaction latency time when peer-to-peer PCI-X bus transactions are occurring and host-to-PCI-X or PCI-X-to-memory transactions are also occurring concurrently therewith.

Host-to-PCI-X transactions, typically, have a higher priority than other PCI-X transactions such as memory or peer-to-peer transactions. In the present invention, it is contemplated that peer-to-peer PCI-X transactions may be enabled or disabled under software control by setting a bit in a register in the core logic 204 (not illustrated). When peer-to-peer PCI-X transactions need to occur, there is a possibility of the peer-to-peer PCI-X transactions being starved due to heavy host-to-PCI-X transactions. The present invention solves this problem by allowing at least one PCI-X-to-PCI-X transaction to occur between host-to-PCI-X transactions. For example, the PCI-X target flow controller 430 (FIG. 4) observes that a PCI-X-to-PCI-X transaction is pending and asserts a processor "retry" on control bus 446 which causes the CPU interface 422 to assert a processor retry signal on the host bus 103. This processor retry signal may be the Backoff (BOFF#) input for 486 and PENTIUM type processors, or a Retry Response (DEFER# asserted with HITM# inactive during the SnooP Phase) for the PENTIUM PRO processors. Other types of processors have similar types of processor retry inputs and are well known to those skilled in the art of microcomputer design. Once the PCI-X-to-PCI-X transaction is underway, the processor "retry" signal is deasserted and another host-to-PCI-X transaction request is allowed to occur. The present invention thus alternates between host-to-PCI-X transactions and PCI-X-to-PCI-X transactions if both are pending.

The PCI-X Specification requires that the PCI-X bridges must follow certain transaction ordering rules to avoid "deadlock" and/or maintain "strong" ordering. To guarantee that the results of one PCI-X initiator's write transactions are observable by other PCI-X initiators in the proper order of occurrence, even though the write transactions may be posted in the PCI-X bridge queues, the following rules must be observed:

- 1) Posted memory writes moving in the same direction through a PCI-X bridge will complete on the destination bus in the same order they complete on the originating bus;
- 2) Write transactions flowing in one direction through a PCI-X bridge have no ordering requirements with respect to write transactions flowing in the other direction of the PCI bridge; and

- 3) Posted memory write buffers in both directions must be flushed or drained before starting another read transaction.

A "PCI-X retry" is defined in the PCI-X Specification as a termination requested by a target before any data is transferred because the target is busy and temporarily unable to process the transaction. The PCI-X target flow controller 430 issues a "PCI-X retry" over control bus 448 during the first transaction data phase if there is M byte aligned address commonality, where $M=16 \times 2^n$ and n is 0, 1, 2, 3, 4, etc., as determined by the PCI-X address comparator 428, thus concurrent data is prevented from being transferred during any M byte aligned address transactions where a write transaction is involved. The PCI-X retry is indicated to the PCI-X device initiator by the respective PCI-X bus interface (412, 414 or 416) asserting Stop ("STOP#") and deasserting Target Ready ("TRDY#) while keeping Device Select ("DEVSEL#") asserted. This tells the PCI-X transaction initiator that the PCI-X target does not intend to complete transaction (TRDY# deasserted) and that the PCI-X transaction initiator must stop the transaction on this data phase (STOP# asserted). The continued assertion of DEVSEL# indicates that the PCI-X transaction initiator must retry the transaction at a later time (when the commonly addressed transaction on the other PCI-X physical bus has completed). Furthermore, the PCI-X transaction initiator must use the exact same address, command and byte enables. If it's a write transaction, it must use exactly the same data in the first data phase. The access must be retried until the transaction is completed. In this way transactions which have M byte aligned addresses and involve a write transaction, therefore should not occur concurrently, are thus easily handled by the core logic 204.

Each PCI-X device embedded on the computer system motherboard, or as a device card inserted into the PCI-X connectors 306, 308, 310, 314 (FIG. 3) require request (REQ#) and grant (GNT#) signals. According to the PCI-X Specification, a PCI-X device is selected and allowed to become the PCI-X bus initiator when it asserts its respective REQ# signal onto the PCI-X bus and the PCI-X arbiter acknowledges the PCI-X device bus initiator request by asserting the respective GNT# signal back to PCI-X device requesting the PCI-X bus. In the core logic 204 of the present invention, a plurality of request and grant signals are available for all of the PCI-X devices in the computer system. The PCI-X arbiter 432, through the respective PCI-X bus interface, may assert a grant signal to another PCI device requesting a transaction before the present PCI-X device transaction is finished as disclosed above.

An additional feature of the present invention is the ability to operate with a "mixed mode" of PCI-X compatible devices and non PCI-X (i.e., PCI) devices. Such a mixed mode configuration is shown in FIG. 4B. Referring to FIG. 4B, the core logic 204 is shown in the center. The host bus 103 and the memory bus 105 are both connected to the core logic 204. As before, the core logic 204 contains the target flow controller 430, an arbiter 432, and an address comparator 428. In addition, one or more configuration memory registers 462 are provided to retain the configuration information regarding the various devices connected to the core logic. This configuration may be determined upon startup (POST) or later by a standard polling mechanism. The configuration information, such as whether a particular device is PCI-X compatible, its attendant address range, and its bandwidth capabilities, are stored in the configuration registers 462. Alternatively, the configuration information can be kept in main memory (system RAM) 106. In the

preferred embodiment of the present invention, a clock 464 is also provided within the core logic 204. However, the run time clock 144 (see FIG. 2) or some other system clock could be used instead.

In FIG. 4B, there are four PCI-X capable busses 206, 208, 210, and 212 with corresponding queues 466, 468, 470, and 472, respectively. Each of the queues 466, 468, 470, and 472 may contain bus read/write queues, a bus interface, and/or transaction address filters (see FIG. 4 and FIG. 4A). The queues 466, 468, 470, and 472 are, in turn, connected to the target flow controller 430 as shown in FIG. 4B. The bridge of the present invention allows a wide variety of device configurations. It will be understood by those skilled in the art that the configuration shown in FIG. 4B is merely illustrative of one of many different configurations made possible by the present invention. For example, as shown in FIG. 4B, a first PCI-X compatible device 476 is connected to bus 206, which operates at 133 MHz. Similarly, a legacy input/output device 482 is connected to bus 212 which, like the device 482, operates at 33 MHz. PCI-X compatible device 480 is connected to bus 201. Finally, bus 208, which operates at 66 MHz, has connected to it two devices, PCI device 478 and PCI-X device 479. Note that, as mentioned previously, in a mixed mode configuration as shown in FIG. 4B, all of the devices behave according to the standard PCI protocol. During configuration (e.g., POST) each of the devices is polled to determine whether or not it is PCI-X compatible. If one or more of the devices is not PCI-X compatible, each of the PCI-X compatible devices is configured to run as a standard PCI device. As the PCI-X protocol is a superset of the standard PCI protocol, each PCI-X compatible device is backward compatible with the standard PCI protocol. According to the PCI-X specification, each PCI-X compliant device must have the capability to be set to operate according to the standard PCI protocol in lieu of the more capable PCI-X protocol. This enables the arbiter 432, in conjunction with the target flow controller 430, to resolve I/O conflicts among PCI and PCI-X devices 476, 478, 479, 490, and 482 because the PCI devices are incapable of participating in arbitration according to the PCI-X protocol in which there is no need to check two devices for contention (wait states). Note that the use of the queues 466, 468, 470, and 472 in the present invention enable the various busses 206, 208, 210, and 212, respectively, to operate at disparate clock rates so that full advantage of each individual device may be achieved.

The present invention, as shown in FIG. 4B, is also capable of handling split bandwidths. Moreover, the present invention is further capable of handling both fixed sized buffers and dynamically allocated buffers. In the preferred embodiment, the core logic 204 would contain sufficient memory for the necessary buffers. However, the buffers containing the queues and/or the transaction address filters of the present invention can be allocated from and located on the system RAM 106. If allocated on the system RAM 106, appropriate allocation calls are made by the target flow controller 260 to the host operating system. While in the preferred embodiment, the host operating system is not involved with transactions. Instead, the host operating system simply "sees" one physical PCI bus. During configuration (e.g., POST), a map is made of the several segments (address ranges) of the various I/O devices connected to the core logic 204. A memory range decoder is included within the core logic 206 which is used by the computer system BIOS 140 to indicate to the host operating system what type of device is in which address range. This process is called bus discovery.

Referring now to FIG. 5, a schematic functional block diagram of an embodiment of a portion of the invention of FIG. 4 is illustrated. For clarity only two of the separate physical PCI-X busses 206, 208 and their respective PCI-X bus interfaces 412, 414 are illustrated. More than two physical PCI-X buses, however, are contemplated herein for the present invention. All PCI-X bus signals, as more fully defined in the PCI-X Specification are connected between the PCI-X busses 206, 208 and their respective PCI bus interfaces 412, 414. A 32-bit address and data bus (AD [31:0]) is illustrated but a 64-bit address and data bus (AD[63:0]) is also contemplated herein as more fully defined in the PCI-X Specification.

For example, a transaction occurring with PCI-X device A (not illustrated) on the PCI-X bus 206 (bus A), generates addresses on the PCI-X bus 206 which are also sent to the PCI-X address comparator 428 (bus A addr). When a transaction is requested (REQ#) by another PCI-X device B (not illustrated) on the PCI-X bus 208 (bus B), a grant (GNT#) is issued by the PCI-X arbiter 432 to the PCI device B. Once the grant is received by the PCI-X device B, the PCI-X device B asserts its transaction address on the PCI-X bus 208. The asserted address from the PCI-X device B is sent to the PCI-X address comparator 428 where the PCI-X device B address is compared with the current transaction address of the PCI-X device A. In order for the PCI-X address comparator 428 to compare the addresses from each one of the PCI-X devices A, B, a wait state is initiated by the PCI-X target flow controller 430 (wait state enable B) to the PCI-X bus interface 414. The PCI-X bus interface 414 causes a wait state to occur by delaying (blocking) assertion of Target Ready (TRDY#) from the target PCI-X device to the PCI-X device B which is the initiator of the new transaction on the PCI-X bus 208. If the compared prefetchable memory addresses are not to the same cache-line, nor are they M byte aligned, where $M=16 \times 2^n$ and n is 0, 1, 2, 3, 4, etc., then the PCI-X bus interface 414 allows assertion of (unblocks) TRDY# from the target PCI-X device and the transaction on the PCI-X bus 208 proceeds to its data phase. Thus, concurrent transactions may occur on the PCI-X busses 206, 208. However, if the compared addresses are the same or are M byte aligned, then the PCI-X target flow controller initiates a PCI-X Retry (retry b) to the PCI-X bus interface 414 which issues a PCI-X Retry to the PCI-X device B. PCI-X Retry need only be asserted when one or both of the PCI-X devices A, B are doing or intend to do write transactions. Since the ordering rules are always "strong" in this embodiment of the present invention, pending write queues are emptied first before starting the next transaction having aligned addresses.

Referring now to FIG. 5A, a schematic functional block diagram of another embodiment of a portion of the invention of FIG. 4 is illustrated. Operation of the embodiment illustrated in FIG. 5A is similar to the embodiment of FIG. 5 except that range registers 436 and transaction queue controller 434 have been added so that when "weak" ordering of memory transactions is appropriate, out-of-order PCI-X transactions such as read transactions bypassing posted write transactions may be performed to reduce latency of the PCI-X bus transactions. It is contemplated in this embodiment that there is at least one pair of range registers for each PCI-X device in the computer system. The at least one pair of range registers holds the upper and lower memory addresses of the respective PCI-X device which require "strong" ordering for that PCI-X device. Any PCI-X memory transactions outside of the "range" of memory addresses defined by the upper and lower memory addresses

stored in the pair of range registers would not require "strong" ordering, thus "weak" ordering may be used to improve bus transaction latency.

The transaction queue controller 434 determines whether a PCI-X device transaction requires "strong" or "weak" ordering by comparing the current and pending transaction addresses in the queues 402, 404 with the corresponding PCI-X device address ranges defined by the range registers 436. When the current and pending transaction addresses do not correspond to those in the range registers 436, the transaction queue controller 434 may instruct the PCI-X target flow controller 430 to advance transactions out of order and/or do read transactions before the write transactions ("weak" ordering) have been flushed from the queues 402, 404. On the other hand, when the current and pending transaction addresses do correspond to those in the range registers 436, the transaction queue controller 434 instructs the PCI-X target flow controller 430 to advance transactions in accordance with the aforementioned PCI-X Specification ordering rules ("strong" ordering).

Referring now to FIG. 5B, a schematic representation of range registers according to the aforementioned embodiment of the present invention is illustrated. Each PCI-X device(x), where x is a to n, has a tag ID register 502 and at least one pair of range registers 504, 506 associated therewith. The lower address range register 506 may contain the lowest memory address of interest to the associated PCI-X device. The range register 506 may be a full 32 or 64-bit register and contain the absolute value lower address. The upper address range register 504 may contain only the offset address of the address stored in the range register 506, i.e., the contents of the range register 504 is added to the contents of the range register 506 to give the absolute value upper memory address of interest to the associated PCI-X device. A plurality of range register pairs also may be associated with a PCI-X device, thus allowing non-contiguous memory address ranges to be programmed for a particular PCI-X device. The computer system startup software during POST, or an API or applications program may also load the range registers 504, 506 with the desired memory address ranges that require strong ordering of PCI-X transactions. The remaining memory addresses which fall outside of the strong ordering address ranges may be treated by the present invention as weak ordering which allows PCI-X transactions to be taken out of order so as to improve PCI-X bus transaction latency times.

Referring to FIG. 6, a schematic state diagram of the present invention is illustrated. Signal conventions hereinafter are the same or similar to those disclosed in Appendix B of the PCI-X Specification incorporated herein by reference. The present invention functions substantially the same as the state machine represented and described in Appendix B of the PCI 2.1 Specification.

Referring now to FIGS. 7, 8 and 9, a process flow diagram of the present invention is illustrated. The aforementioned process flow diagram describes the operation of preferred embodiments of the present invention. In step 702, the present invention detects a PCI-X bus transaction(s) and in step 704 asserts a wait state. The step 704 wait state is used to allow sufficient time for comparison of a current PCI-X transaction address with a new (pending) PCI-X transaction address in step 706. During the address comparison in step 706, the present invention determines from the pending transaction address whether the pending transaction is a peer-to-peer (decision step 708), an I/O cycle (decision step 710), or a prefetchable memory (decision step 712) transaction, and whether there is an address match or M byte

aligned, where $M=16 \times 2^n$ and n is 0, 1, 2, 3, 4, etc., address commonality (decision step 714).

Host-to-PCI-X transactions, typically, have a higher priority of execution than either PCI-X-to-memory or PCI-X-to-PCI-X transactions. The logic of the present invention will alternate (flip-flop) between execution of a host-to-PCI-X transaction and a peer-to-peer PCI-X transaction so that the peer-to-peer PCI-X transactions are not "starved" by the higher priority host-to-PCI-X transactions. Decision step 708 determines whether there is a peer-to-peer PCI-X transaction pending. Decision step 732 determines if the destination PCI-X bus required by the pending peer-to-peer PCI-X transaction is being used by another PCI-X bus master, i.e., a PCI-X transaction is occurring on the destination PCI-X bus needed for the pending peer-to-peer PCI-X transaction. If the destination PCI-X bus is not being used, then decision step 734 determines if there is a pending host-to-PCI-X transaction. If there is a pending host-to-PCI-X transaction, then decision step 736 determines if a priority status bit is set to one. The priority status bit may be a bit in a status register in the core logic of the present invention and may be used as a one bit flip-flop register to indicate the last type of PCI-X transaction to occur, i.e., execution of a host-to-PCI-X transaction sets the status bit to "one" and execution of a peer-to-peer PCI-X transaction resets the status bit to "zero." Thus, by reading this status bit, the present invention may determine at any time whether a host-to-PCI-X transaction or a peer-to-peer PCI-X transaction occurred last.

If the decision step 736 determines that the status bit is not "one" then a peer-to-peer PCI-X transaction occurred last, and the pending host-to-PCI-X transaction should execute next. This is accomplished in step 738 by asserting a PCI-X retry to the peer-to-peer PCI-X initiator which causes this PCI-X initiator to drop its PCI-X bus request and retry the same request later. In step 740, the host-to-PCI-X transaction proceeds to execution, and in step 742, the status bit is set to one" so that the next peer-to-peer PCI-X transaction request will be given priority over a host-to-PCI-X transaction request.

If the decision step 736 determines that the status bit is set to "one" then a host-to-PCI-X transaction occurred last, and the pending peer-to-peer PCI-X transaction should execute next. This is accomplished in step 744 by asserting a retry to the host processor which causes the host processor to drop its PCI-X bus request and retry the same request later. In step 746, the peer-to-peer PCI-X transaction proceeds to execution, and in step 748, the status bit is set to "zero" so that the next host-to-PCI-X transaction request will be given priority over a peer-to-peer PCI-X transaction request. Step 750 deasserts the retry to the host processor.

In decision step 732, if the required destination PCI-X bus is being used for another PCI-X transaction, then a retry signal is asserted to the pending peer-to-peer PCI initiator in step 738. This causes the pending peer-to-peer PCI-X initiator to retry its transaction request later. If there is a host-to-PCI-X transaction pending, step 740 allows the host-to-PCI-X transaction to proceed, and step 742 will then set the priority status bit to one as described above.

The decision step 710 determines whether the pending PCI-X transaction is a memory address or an I/O address. If a memory address, decision step 712 determines whether the pending PCI-X transaction is a prefetchable memory address. Decision step 714 then determines whether the pending and current PCI-X transactions are accessing the same cache-line or have M byte aligned addresses, where $M=16 \times 2^n$ and n is 0, 1, 2, 3, 4, etc.,. If there is no address

match or alignment of the current and pending PCI-X transactions, then decision step 716 determines whether the pending PCI-X transaction requires weak or strong ordering. In step 718, strong ordering requires that all current posted writes must be flushed before a read transaction may proceed, in accordance with the PCI-X Specification. In step 720, weak ordering allows read transactions to bypass current posted write transactions which may improve PCI-X bus transaction latency times.

If step 710 determines that the pending PCI-X transaction is an I/O cycle, step 712 determines that the pending PCI-X transaction is not a prefetchable memory address, or step 714 determines that the pending and current PCI-X transactions are accessing the same cache-line or have M byte aligned addresses, where $M=16 \times 2^n$ and n is 0, 1, 2, 3, 4, etc., then step 722 allows only one PCI-X transaction to occur. Decision step 724 determines whether more than one PCI-X transaction is pending and if so, step 728 issues a retry to the PCI-X device making the latest transaction request, thus forcing this PCI-X device to retry its request at a later time. When only one PCI-X transaction is pending, step 726 lets that single pending PCI-X transaction to wait by not asserting its GNT# signal until the current PCI-X transaction has finished. When the current PCI-X transaction is finished, the GNT# signal is asserted so that the pending PCI-X transaction may execute in step 730.

Referring now to FIG. 10, a process flow diagram of the arbiter of the present invention is illustrated. A PCI-X transaction request, REQ#, is detected in step 1002 and decision step 1004 determines whether the PCI-X buses are idle or not. If the PCI-X buses are idle, then a grant signal, GNT#, is returned in step 1010 to the requesting PCI-X device. If any of the PCI-X buses are not idle, i.e., there is current PCI-X transaction in progress, then decision step 1006 determines if pipelined PCI-X transactions are allowed. Decision step 1008 determines whether the current and pending transactions are on the same or different physical PCI-X buses.

If step 1006 determines that pipelined PCI-X transactions are allowed and step 1008 determines that the current and pending PCI-X transactions will be on different physical PCI-X buses, the step 1010 allows the grant signal, GNT#, to be returned to the PCI-X device requesting a pending PCI-X transaction. Otherwise, GNT# is not returned to the requesting PCI-X device until the current PCI-X transaction has completed.

The present invention, therefore, is well adapted to carry out the objects and attain the ends and advantages mentioned, as well as others inherent therein. While the present invention has been depicted, described, and is defined by reference to particular preferred embodiments of the invention, such references do not imply a limitation on the invention, and no such limitation is to be inferred. The invention is capable of considerable modification, alternation, and equivalents in form and function, as will occur to those ordinarily skilled in the pertinent arts. The depicted and described preferred embodiments of the invention are exemplary only, and are not exhaustive of the scope of the invention. Consequently, the invention is intended to be limited only by the spirit and scope of the appended claims, giving full cognizance to equivalents in all respects.

What is claimed is:

1. A computer system having a core logic chip set capable of bridging between a processor host bus, memory bus and a plurality of registered peripheral component interconnect (PCI-X) buses wherein the plurality of PCI-X buses each have the same logical PCI-X bus number, said system comprising:

- a central processing unit connected to a host bus;
- a random access memory connected to a random access memory bus;
- a core logic chip set connected to the host bus and the random access memory bus;
- said core logic chip set configured as a first interface bridge between the host bus and the random access memory bus;
- said core logic chip set configured as a plurality of second interface bridges between the host bus and a plurality of PCI-X buses;
- said core logic chip set configured as a plurality of third interface bridges between the memory bus and the plurality of PCI-X buses, wherein the plurality of PCI-X buses are physically separate but have the same logical PCI-X bus number;
- a PCI-X address comparator;
- a PCI-X arbiter for receiving request signals from and issuing grant signals to PCI-X devices connected to said plurality of PCI-X buses; and
- a PCI-X target flow controller;
- said PCI-X address comparator receiving transaction addresses from said plurality of PCI-X bus interfaces, wherein the transaction addresses are compared and an address match is found if the transaction addresses from two or more of said plurality of PCI-X bus interfaces are the same or are within M bytes of each other, where $M=16 \times 2^n$ and n is zero or a positive integer number, then said PCI-X address comparator sends an address match signal to said PCI-X target flow controller which causes a retry signal to be issued from the one of said plurality of PCI-X bus interfaces that corresponds to the newest transaction request causing the address match, if the transaction addresses from two or more of said plurality of PCI-X bus interfaces are not the same nor are the transaction addresses within M bytes then no address match signal is generated.
- 2. The computer system of claim 1, wherein the central processing unit is a plurality of central processing units.
- 3. The computer system of claim 1, wherein the core logic chip set is at least one integrated circuit.
- 4. The computer system of claim 3, wherein the at least one integrated circuit core logic chip set is at least one application specific integrated circuit.
- 5. The computer system of claim 3, wherein the at least one integrated circuit core logic chip set is at least one programmable logic array integrated circuit.
- 6. The computer system of claim 1, further comprising at least one registered peripheral component interconnect (PCI-X) device, the at least one registered peripheral component interconnect device connected to at least one of the plurality of PCI-X buses.
- 7. The computer system of claim 6, wherein the at least one registered peripheral component interconnect device is at least one 32-bit data width registered peripheral component interconnect device.
- 8. The computer system of claim 6, wherein the at least one registered peripheral component interconnect device is at least one 64-bit data width registered peripheral component interconnect device.
- 9. The computer system of claim 6, wherein the at least one registered peripheral component interconnect device runs at a clock of 66 MHz.
- 10. The computer system of claim 6, further comprising a plurality of address range register pairs, wherein a first one

27

of the pair contains a lower memory address and a second one of the pair contains an upper memory address for each of the at least one PCI-X devices in the computer system.

11. The computer system of claim 10, further comprising a transaction queue controller, said transaction queue controller comparing current and pending memory transaction addresses with the addresses represented by said plurality of address range register pairs so that when a match is found between the current and pending memory transaction addresses and those addresses represented in said plurality of address range register pairs then strong ordering is used for transaction execution and when a match is not found then weak ordering is used for transaction execution.

12. The computer system of claim 1, wherein said plurality of PCI-X bus interfaces are configured for 32-bit address and data information.

13. The computer system of claim 1, wherein said plurality of PCI-X bus interfaces are configured for 64-bit address and data information.

14. The computer system of claim 1, further comprising transaction address filters for each of the plurality of PCI-X buses, wherein only PCI-X transactions addressed to a PCI-X device connected to a one of the plurality of PCI-X buses is allowed to be broadcast on the one of the plurality of PCI-X buses.

15. The computer system of claim 1, further comprising said core logic chip set configured as a plurality of fourth interface bridges between the plurality of PCI-X buses for peer-to-peer PCI-X transactions.

16. The computer system of claim 15, wherein a peer-to-peer PCI-X transaction is broadcast on only a one of the plurality of PCI-X buses which is connected to a target PCI-X device.

17. The computer system of claim 15, wherein a peer-to-peer PCI-X transaction is broadcast to a PCI-X device connected to a one of the plurality of PCI-X buses through a transaction filter which allows the PCI-X transaction to be broadcast only on the one of the plurality of PCI-X buses.

18. The computer system of claim 15, wherein said plurality of fourth interface bridges is enabled or disabled by setting or clearing a bit in a configuration register of said core logic chip set.

19. The computer system of claim 15, further comprising logic for asserting a first retry signal on the host bus when a peer-to-peer PCI-X transaction is pending.

20. The computer system of claim 19, further comprising logic for determining whether a current transaction on a PCI-X bus is a first host-to-PCI-X transaction or a first peer-to-peer PCI-X transaction, if the current transaction is the first host-to-PCI-X transaction then the first retry signal is asserted on the host bus when a second peer-to-peer PCI-X transaction and a second host-to-PCI-X transaction are pending so that the pending second peer-to-peer PCI-X transaction executes before the pending second host-to-PCI-X transaction, if the current transaction is the first peer-to-peer PCI-X transaction then a second retry signal is asserted on the PCI-X bus when the second peer-to-peer PCI-X transaction and the second host-to-PCI-X transaction are pending so that the pending second host-to-PCI-X transaction executes before the pending second peer-to-peer PCI-X transaction.

21. The computer system of claim 1, wherein the host bus, random access memory bus, and the plurality of PCI-X buses are on a computer system printed circuit board.

22. The computer system of claim 21, further comprising at least one set of two PCI-X connectors, each set of PCI-X connectors connected to at least one of the plurality of PCI-X buses.

28

23. The computer system of claim 21, further comprising a plurality of sets of two PCI-X connectors each one of the plurality of sets connected to one of the plurality of PCI-X buses.

24. The computer system of claim 21, wherein at least one PCI-X connector is mounted on the printed circuit board and connected to one of the plurality of PCI-X buses.

25. The computer system of claim 1, further comprising said PCI-X target flow controller generating a wait state signal during comparison of the transaction addresses in said PCI-X address comparator.

26. The computer system of claim 1, wherein the PCI-X target flow controller does not cause the retry signal to be issued if the transaction addresses are only for read transactions.

27. The computer system of claim 1, wherein the write transactions stored in said plurality of PCI-X bus read/write queues are flushed before starting the next transaction.

28. The computer system of claim 1, wherein a plurality of PCI-X bus transactions may run concurrently between said plurality of PCI-X bus read/write queues and said random access memory read/write queues.

29. The computer system of claim 1, wherein said PCI-X arbiter may issue a grant signal to a requesting PCI-X device before releasing a grant signal to another PCI-X device doing a current transaction.

30. A method, in a computer system having a core logic chip set capable of bridging between a processor host bus, memory bus and a plurality of registered peripheral component interconnect (PCI-X) buses wherein the plurality of PCI-X buses each have the same logical PCI bus number, said method comprising the steps of:

providing a central processing unit connected to a host bus;

providing a random access memory connected to a random access memory bus;

providing a core logic chip set connected to the host bus and the random access memory bus;

configuring said core logic chip set as a first interface bridge between the host bus and the random access memory bus;

configuring said core logic chip set as a plurality of second interface bridges between the host bus and a plurality of registered peripheral component interconnect (PCI-X) buses, wherein the plurality of PCI-X buses are physically separate but have the same logical PCI-X bus number;

configuring said core logic chip set as a plurality of third interface bridges between the random access memory bus and the plurality of PCI-X buses;

providing a PCI-X address comparator;

providing a PCI-X arbiter for receiving request signals from and issuing grant signals to PCI-X devices connected to said plurality of PCI-X buses; and

providing a PCI-X target flow controller;

said PCI-X address comparator receiving transaction addresses from said plurality of PCI-X bus interfaces, wherein the transaction addresses are compared and an address match is found if the transaction addresses from two or more of said plurality of PCI-X bus interfaces are the same or are within M bytes of each other, where $M=16 \times 2^n$ and n is zero or a positive integer number, then said PCI-X address comparator sends an address match signal to said PCI-X target flow controller which causes a retry signal to be issued from

29

the one of said plurality of PCI-X bus interfaces that corresponds to the newest transaction request causing the address match, if the transaction addresses from two or more of said plurality of PCI-X bus interfaces are not the same nor are the transaction addresses within M bytes then no address match signal is generated.

31. The method of claim 30, further comprising the step of generating a wait state signal during comparison of the transaction addresses in said PCI-X address comparator.

32. The method of claim 30, wherein the PCI-X target flow controller does not cause the retry signal to be issued if the transaction addresses are only for read transactions.

33. The method of claim 30, wherein peer-to-peer PCI-X bus transactions occur between said plurality of PCI-X bus read/write queues.

34. The method of claim 30, further comprising the steps of:

storing a lower memory address in a lower range register associated with a PCI-X device;

storing an upper memory address in an upper range register associated with the PCI-X device;

comparing a transaction memory addresses with a range of addresses between the stored lower and upper memory addresses;

using strong ordering for PCI-X transactions when the transaction memory addresses thereof are found within the range of addresses; and

using weak ordering for PCI-X transactions when the transaction memory addresses thereof are not found within the range of addresses.

35. The method of claim 30, further comprising the steps of:

storing memory and I/O addresses, associated with each PCI-X device connected to a one of the plurality of PCI-X buses, in a plurality of transaction address filter registers, each of the plurality of transaction address filter registers associated with a respective one of the plurality of PCI-X buses; and

comparing a PCI-X transaction address with the stored memory and I/O addresses in the plurality of transaction address filter registers to determine which one of the respective one of the plurality of PCI-X buses the PCI-X transaction address should be broadcast on.

36. A method, in a computer system having a core logic chip set capable of bridging between a processor host bus, memory bus and a plurality of registered peripheral component interconnect (PCI-X) buses wherein the plurality of PCI-X buses each have the same logical PCI-X bus number, said method comprising the steps of:

providing a central processing unit connected to a host bus;

providing a random access memory connected to a random access memory bus;

providing a core logic chip set connected to the host bus and the random access memory bus;

configuring said core logic chip set as a first interface bridge between the host bus and the random access memory bus;

configuring said core logic chip set as a plurality of second interface bridges between the host bus and a plurality of registered peripheral component interconnect (PCI-X) buses, wherein the plurality of PCI-X buses are physically separate but have the same logical PCI-X bus number;

30

configuring said core logic chip set as a plurality of third interface bridges between the random access memory bus and the plurality of PCI-X buses;

configuring said core logic chip set as a plurality of fourth interface bridges between the plurality of PCI buses

providing a PCI-X address comparator;

providing a PCI-X arbiter for receiving request signals from and issuing grant signals to PCI-X devices connected to said plurality of PCI-X buses; and

providing a PCI-X target flow controller;

said PCI-X address comparator receiving transaction addresses from said plurality of PCI-X bus interfaces, wherein the transaction addresses are compared and an address match is found if the transaction addresses from two or more of said plurality of PCI-X bus interfaces are the same or are within M bytes of each other, where $M=16 \times 2^n$ and n is zero or a positive integer number, then said PCI-X address comparator sends an address match signal to said PCI-X target flow controller which causes a retry signal to be issued from the one of said plurality of PCI-X bus interfaces that corresponds to the newest transaction request causing the address match, if the transaction addresses from two or more of said plurality of PCI-X bus interfaces are not the same nor are the transaction addresses within M bytes then no address match signal is generated.

37. The method of claim 36, wherein comparing transaction addresses comprises the steps of:

detecting a first pending PCI-X transaction having a second address by a PCI-X device asserting a transaction request on a one of the plurality of PCI-X buses; asserting a wait state to the PCI-X device; and

comparing the first pending PCI-X transaction second address with a current PCI-X transaction having a first address, comprises the steps of:

determining if the first pending PCI-X transaction second address is for an input-output (I/O) address then waiting for the current PCI-X transaction to complete before granting the PCI-X device transaction request;

determining if the first pending regPCI transaction second address is not a prefetchable memory address then waiting for the current PCI-X transaction to complete before granting the PCI-X device transaction request;

determining if the first pending PCI-X transaction second address is the same or is within M bytes of the current PCI-X transaction first address, where $M=16 \times 2^n$ and n is zero or a positive integer number, then waiting for the current PCI-X transaction to complete before granting the PCI-X device transaction request;

otherwise, granting the first pending PCI-X device transaction request before the current PCI-X transaction has completed.

38. The method of claim 37, wherein the step of comparing the first pending transaction second address with a current PCI-X transaction first address further comprises the steps of:

determining if the first pending PCI-X transaction is a peer-to-peer PCI-X transaction, wherein if the first pending PCI-X transaction is a peer-to-peer PCI transaction, further comprising the steps of: determining if the current PCI-X transaction is a peer-to-peer PCI-X transaction;

31

determining if the current PCI-X transaction is a host-to-PCI-X transaction;
 asserting a first retry signal to the PCI-X device requesting the first pending PCI-X transaction if the current PCI-X transaction is a peer-to-peer transaction and a second pending PCI-X transaction is a host-to-PCI-X transaction; otherwise,
 asserting a second retry signal to the central processing unit connected to the host bus if the current PCI-X transaction is a host-to-PCI-X transaction and the second pending PCI-X transaction is a host-to-PCI-X transaction;
 granting the first pending PCI-X device transaction request; and
 deasserting the second retry signal to the central processing unit.

39. The method of claim 37, further comprising the steps of:

determining if more than one PCI-X transaction request is pending, wherein:
 if only one PCI-X transaction is pending assert a wait to the PCI-X device until the current PCI-X transaction is finished then grant the PCI-X device transaction request; and
 if more than one PCI-X transaction is pending then issue a retry to the last PCI-X device asserting a transaction request.

40. The method of claim 36, wherein operation of the PCI-X arbiter comprises the steps of:

detecting a PCI-X device bus request;
 determining if the plurality of PCI-X buses are idle, if so then issuing a grant to the PCI-X device;
 determining if concurrent PCI-X device transactions are permitted, if so then issuing a grant to the PCI-X device; and
 determining if the PCI-X device bus request is on a PCI-X bus that is idle, if so then issuing a grant to the PCI-X device; otherwise,
 waiting until the plurality of PCI-X buses are idle before issuing a grant to the PCI-X device.

41. A core logic chip set capable of bridging between a processor host bus, memory bus and a plurality of registered peripheral component interconnect (PCI-X) buses wherein the plurality of PCI-X buses each have the same logical PCI-X bus number, comprising:

a plurality of PCI-X bus read/write queues;
 a plurality of PCI-X bus interfaces adapted for connection to a plurality of PCI-X buses;
 said plurality of PCI-X bus read/write queues connected to said plurality of PCI-X bus interfaces, wherein read and write transactions through said plurality of PCI-X bus interfaces are stored in said plurality of PCI-X bus read/write queues;
 processor read/write queues;
 a processor interface connected to said processor read/write queues, said processor interface adapted for connection to a processor host bus;
 random access memory read/write queues;
 a random access memory interface connected to said random access memory read/write queues, said random access memory interface adapted for connection to a random access memory bus;
 said random access memory queues connected to said processor read/write queues;
 said plurality of PCI-X bus read/write queues connected to said random access memory queues;

32

said plurality of PCI-X bus read/write queues connected to said processor read/write queues;

a PCI-X address comparator;

a PCI-X arbiter adapted for receiving request signals from and issuing grant signals to PCI-X devices connected to said plurality of PCI-X buses; and

a PCI-X target flow controller;

said PCI-X address comparator adapted to receive transaction addresses from said plurality of PCI-X bus interfaces, wherein the transaction addresses are compared and an address match is found if the transaction addresses from two or more of said plurality of PCI-X bus interfaces are the same or are within M bytes of each other, where $M=16 \times 2^n$ n is zero or a positive integer number, then said PCI-X address comparator sends an address match signal to said PCI-X target flow controller which causes a retry signal to be issued from the one of said plurality of PCI bus interfaces that corresponds to the newest transaction request causing the address match, if the transaction addresses from two or more of said plurality of PCI-X bus interfaces are not the same nor are the transaction addresses within M bytes then no address match signal is generated.

42. The core logic chip set according to claim 41, further comprising said PCI-X target flow controller generating a wait state signal during comparison of the transaction addresses in said PCI-X address comparator.

43. The core logic chip set according to claim 41, wherein the PCI-X target flow controller does not cause the retry signal to be issued if the transaction addresses are only for read transactions.

44. The core logic chip set according to claim 41, wherein the write transactions stored in said plurality of PCI-X bus read/write queues are flushed before starting the next transaction.

45. The core logic chip set according to claim 41, wherein a plurality of PCI-X bus transactions may run concurrently between said plurality of PCI-X bus read/write queues and said random access memory read/write queues.

46. The core logic chip set according to claim 41, wherein said PCI-X arbiter may issue a grant signal to a requesting PCI-X device before releasing a grant signal to another PCI-X device doing a current transaction.

47. The core logic chip set according to claim 41, wherein said plurality of PCI-X bus interfaces are configured for 66 megahertz (MHz) operation.

48. The core logic chip set according to claim 41, wherein said plurality of PCI-X bus interfaces are adapted for 32-bit address and data information.

49. The core logic chip set according to claim 41, wherein said plurality of PCI-X bus interfaces are adapted for 64-bit address and data information.

50. The core logic chip set according to claim 41, wherein said plurality of PCI-X bus read/write queues are adapted for peer-to-peer PCI-X bus transactions occurring therebetween.

51. The core logic chip set according to claim 41, further comprising:

a plurality of range register pairs, said plurality of range register pairs adapted to provide a pair of range registers for each one of a plurality of PCI-X devices;

said plurality of range registers storing upper and lower memory addresses; and

a transaction queue controller, said transaction queue controller adapted for comparing PCI-X transaction

33

addresses with addresses between the stored upper and lower memory addresses, when there is a comparison match said transaction queue controller causes strong ordering of PCI-X transactions, and when there is not a comparison match said transaction queue controller causes weak ordering of PCI-X transactions.

52. The core logic chip set according to claim 41, further comprising:

transaction address filters connected to each of said plurality of PCI-X bus read/write queues, wherein only transactions for an intended PCI-X device reaches an associated one of said plurality of PCI-X bus read/write queues.

53. The core logic chip set according to claim 41, further comprising:

transaction address filters connected to each of said plurality of PCI-X bus read/write queues, wherein only transactions for an intended PCI-X device reaches an associated one of said plurality of PCI-X bus read/write queues.

54. The core logic chip set according to claim 41, further comprising:

a host processor retry logic adapted for delaying a next host-to-PCI-X transaction until a pending peer-to-peer PCI-X transaction has executed.

55. The core logic chip set according to claim 54, further comprising logic adapted for determining whether a current transaction on a PCI-X bus is a first host-to-PCI-X transaction or a first peer-to-peer PCI transaction, if the current transaction is the first host-to-PCI-X transaction then the first retry signal is asserted on the host bus when a second peer-to-peer PCI-X transaction and a second host-to-PCI-X transaction are pending so that the pending second peer-to-peer PCI-X transaction executes before the pending second host-to-PCI-X transaction, if the current transaction is the first peer-to-peer PCI-X transaction then a second retry signal is asserted on the PCI-X bus when the second peer-to-peer PCI-X transaction and the second host-to-PCI-X transaction are pending so that the pending second host-to-PCI transaction executes before the pending second peer-to-peer PCI-X transaction.

56. A computer system having a core logic chip set capable of bridging between a processor host bus, memory bus and at least two registered peripheral component interconnect (PCI-X) compatible buses, each of said buses having an interface, said buses connecting at least one input/output device, said input/output device being either a PCI-X compatible device or a PCI compatible device, each of said buses having an identical logical bus number, said system comprising:

a central processing unit connected to a host bus;

a random access memory connected to a random access memory bus;

a core logic chip set connected to the host bus and the random access memory bus;

said core logic chip set configured as a first interface bridge between said host bus and said random access memory bus;

said core logic chip set further configured as a plurality of second interface bridges between said host bus and said PCI-X buses;

said core logic chip set further configured as a plurality of third interface bridges between said memory bus and said buses, said PCI-X buses are physically separate;

an address comparator;

an arbiter for receiving request signals from and issuing grant signals to said input/output devices connected to said PCI-X buses; and

34

a target flow controller;

said address comparator receiving transaction addresses from said at least two PCI-X bus interfaces, wherein the transaction addresses are compared and an address match is found if the transaction addresses from two or more of said plurality of PCI-X bus interfaces are the same or are within M bytes of each other, where $M=16 \times 2^n$ and n is zero or a positive integer number, then said address comparator sends an address match signal to said target flow controller which causes a retry signal to be issued from the one of said at least two bus interfaces that corresponds to the newest transaction request causing the address match, if the transaction addresses from two or more of said at least two bus interfaces are not the same nor are the transaction addresses within M bytes then no address match signal is generated.

57. A computer system having a core logic chip set capable of bridging between a processor host bus, memory bus and at least one registered peripheral component interconnect (PCI-X) compatible bus and at least one PCI compatible bus, each of said buses having an interface, said buses connecting at least one input/output device, said input/output device being either a PCI-X compatible device or a PCI compatible device, said at least two buses each having the same logical bus number, said system comprising:

a central processing unit connected to a host bus;

a random access memory connected to a random access memory bus;

a core logic chip set connected to the host bus and the random access memory bus;

said core logic chip set configured as a first interface bridge between said host bus and said random access memory bus;

said core logic chip set further configured as a plurality of second interface bridges between said host bus and said at least two buses;

said core logic chip set further configured as a plurality of third interface bridges between said memory bus and said at least two buses, said at least two buses are physically separate but have an identical logical bus number;

an address comparator;

an arbiter for receiving request signals from and issuing grant signals to said input/output devices connected to said at least two buses; and

a target flow controller;

said address comparator receiving transaction addresses from said at least two bus interfaces, wherein the transaction addresses are compared and an address match is found if the transaction addresses from two or more of said bus interfaces are the same or are within M bytes of each other, where $M=16 \times 2^n$ and n is zero or a positive integer number, then said address comparator sends an address match signal to said target flow controller which causes a retry signal to be issued from the one of said at least two bus interfaces that corresponds to the newest transaction request causing the address match, if the transaction addresses from two or more of said bus interfaces are not the same nor are the transaction addresses within M bytes then no address match signal is generated.

* * * * *